COMBINATORIAL STRUCTURES AND FILTER DESIGN IN INFORMATION SPACES

BY

JINGJIN YU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Daniel Liberzon, Chair
Professor Seth Hutchinson
Professor Steven M. LaValle
Assistant Professor Sayan Mitra

ABSTRACT

In this thesis, we develop a filtering process called combinatorial filters for handling combinatorial processes that evolve over time and study two practical problems using this method.

The first problem is a generalization of the sensing aspect of visibility-based pursuit evasion games, in which the task is to maintain the distribution of hidden targets that move outside the field of view while a sensor sweep is being performed. For this problem, we apply information space concepts to significantly reduce the general complexity so that information is processed only when the shadow region (all points invisible to the sensors) changes combinatorially or targets pass in and out of the field of view. The cases of distinguishable, partially distinguishable, and completely indistinguishable targets are handled. Depending on whether the targets move nondeterministically or probabilistically, more specific classes of problems are formulated. For each case, efficient filtering algorithms are introduced, implemented, and demonstrated that provide critical information for tasks such as counting, herding, pursuit-evasion, and situational awareness.

Next, we study the problem of using sparse, heterogeneous sensor data to verify the stories (i.e., path samples) of agents. Since there are two sets of data, the combinatorial filter for this problem can be built in two ways: Using a filter (an automaton) built from sensor data to process the story or using a filter built from the story to process the sensor data. Both approaches lead to dynamic programming based efficient algorithms for extracting a compatible path if one exists. In addition to exact path inference, our method also applies to approximate path inference that allows errors in data. Besides immediate applicability toward security and forensics problems, the idea of behavior validation using external sensors also appears promising in complementing design time model verification.

To Yunxia and Mandy

ACKNOWLEDGMENTS

The result presented here would not have been possible without the support of many people. The ultimate honor goes to my dear wife, Yunxia, who endured this seemingly endless process with me while I, irresponsible as a husband, indulged myself in the playful matters of mathematics, robotics, and control theory. Many thanks to my advisor, Steven M. LaValle, whose sole effort upon his students is to make them genuine researchers in every aspect. Besides Steve, I would also like to thank others members of my doctoral thesis committee, Seth Hutchinson, Daniel Liberzon, and Sayan Mitra, for their invaluable inputs into my thesis research during the past few years. In particular, I would like to extend my appreciation for the extraordinary help I received from Daniel, who is a great teacher, collaborator, and in many ways also an advisor. And finally, thanks go to my academic siblings (Jason O'Kane, Stephen Lindemann, Hamidreza Chitsaz, Anna Yershova, Benjamin Tovar, Aneel Tanwani, Lars Erickson, Kamilah Taylor, Leonardo Bobadilla, Dmitry Yershov, Oscar Sanchez, and Max Katsev) and numerous others who enjoyed and suffered this long process with me, always offering help, support, and perhaps more importantly, constructive criticism, all of which have made me a better human being.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

A few decades have passed since the emergence of robotics as an independent research field. Over the past thirty or so years, we see a great number of fundamental and challenging problems in robotics being tackled and solved with great success [1, 2, 3]. In addition to elegant mathematical solutions, many powerful algorithms and tools were produced, such as *cylindrical algebraic decomposition* [4, 5], *Canny's roadmap algorithm* [6], *PRM* [7], *RRT* [8, 9], and *SLAM* [10], to name a few.

The resolution of these fundamental problems also brought a dramatic increase in robotics applications in recent years. Due to advancements in areas such as robot control, sensor modeling, grasping, manipulation, and motion planning, robots are now routinely employed to help with a diversity of tasks that requires a high degree of autonomy and reliability, such as ex-terrestrial exploration and driverless navigation. In the Mars rover project by NASA, the rover Opportunity's operating hours already exceed its designed expectation (90 Martian days) by 30 times[1,2] and is still going strong as this thesis is being submitted. As another example, merely ten years ago, the idea of having an automobile driving itself in regular traffic lived largely in the realm of science fiction. After all, it is hard to imagine that a rational being would willingly sit in a driverless car going at sixty miles per hour, relaxed, knowing that his or her life is completely in the control of a computer algorithm. Yet unbeknown to the most of us at that time, this was set to change shortly. On October 8, 2005, five driverless vehicles completed the second DARPA Grand Challenge, which required the vehicles to

[1] `http://en.wikipedia.org/wiki/Opportunity_rover`
[2] `http://www.nasa.gov/mission_pages/mer/news/mer20100519.html`

navigate a 132-mile off-road course in a Nevada desert [11, 12]. Then, on November 3, 2007, six autonomous vehicles successfully finished the DARPA Urban Challenge (the third DARPA Grand Challenge) in a simulated urban driving environment [13, 14, 15]. More recently, Google announced[3] that their driverless cars, driven in regular traffic, had logged 1,000 miles without any human intervention and over 140,000 miles with occasional human intervention. Suddenly, the reality of having automobiles driving themselves appears quite reachable. These examples provide concrete evidence that system autonomy and robustness continue to break barriers, making what was once unimaginable within grasp.

The advancement in robotics also brought proliferation and commercial success of household robots. Even though industrial robots have long been used in assembly lines for improved efficiency and accuracy, the presence of robots in the typical household had been very limited. This scenario, due partly to the once prohibitively high cost of parts and partly to the complexity of household environment, is rapidly changing. During the past decade, the continuously dropping price and shrinking size of mechanical, sensing, and computing components have finally broken the usability versus price barrier and robots are entering our homes. One prominent example is the iRobot[4] Roomba (see Fig. 1.1(d)): They are designed to clean the floor autonomously, blanketing its service area using seemingly clever coverage algorithms. To date, more than 6 million home robots made by iRobot have been sold worldwide.[5] Other examples of robots in/around the household include RoboMower (see Fig. 1.1(e)) from Friendly Robotics,[6] RoboMop[7] (see Fig. 1.1(f)), and many others.[8] As the downward trend in component size and cost accelerates, we can expect the further proliferation of robots into households, doing our chores and entertaining us.

[3] http://googleblog.blogspot.com/2010/10/what-were-driving-at.html

[4] http://www.irobot.com/

[5] http://seekingalpha.com/article/252090-irobot-ceo-discusses-q4-2010-results-earnings-call-transcript

[6] http://friendlyrobotics.ca

[7] http://www.robomop.net/

[8] http://en.wikipedia.org/wiki/Domestic_robot

(a)

(b)

(c)

(d)

(e)

(f)

Figure 1.1: Advanced robotics applications and household robots. (a) Stanley, the winning entry of 2005 Darpa Grand Challenge from Stanford University. (b) Boss, the winning entry of 2007 Darpa Urban Challenge from Carnegie Mellon University. (c) The Toyota Prius version of the Google driverless car. (d) iRobot Roomba vacuuming robot. (e) RoboMower from Friendly Robotics. (f) RoboMop.

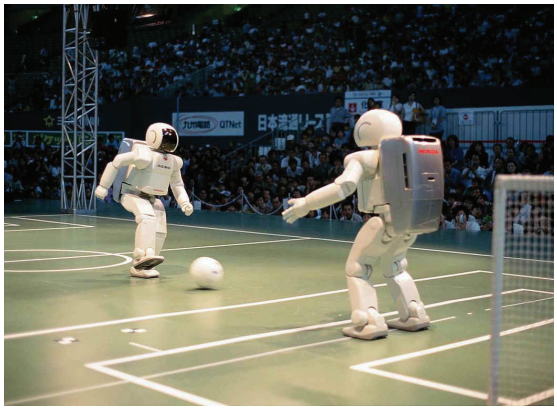
1.1 Some Current Challenges

The rapid development in robotics theory and application also brought us a fresh round of challenges. At present, a high degree of automation is realized in only a handful of complicated but specialized applications – we are far from understanding and reaching full autonomy with which general purpose robots can be built with ease. What are the roadblocks that slow us down? Roughly speaking, any robot can be viewed as an (possibly hierarchical) integration of sensing, computation, and actuation (control) components. What distinguishes robots from many other automation apparatuses such as computers is that robots must *interact* with the physical world. Since the physical world is highly unstructured and therefore unpredictable, there can be a high degree of inherent complexity and uncertainty, presenting a major and unique challenge for the designers of robots (Here, *structure*, as a criterion for classifying robotics problems, refers to predictability, prearrangement, and organization [16]). The problem is further complicated by the fact that robots must also interact with other robots as well as humans.

Such challenges in dealing with uncertainty are quite evident in the development of robotics applications. Since uncertainty is hard to cope with, the natural approach to limit uncertainty gave rise to the idea of tightly controlling the environment with which the robots interact. Although this idea was far from original (e.g. sewing machines, railways for running trains, or even plain old wheels), adapting it under a robotics context lead to dramatically improved productivity (e.g. robot arms for assembling automobiles in factories). On the other hand, such robots and machines are highly specialized, which ultimately restricts their usefulness to environments for which they are designed, and nowhere else. This can be highly undesirable at times. For example, if an assembly line can only produce a single product, a change in the product design will require the entire assembly line to be updated. The development of *Flexible Manufacturing Systems*[9] partially addressed this issue, adding some

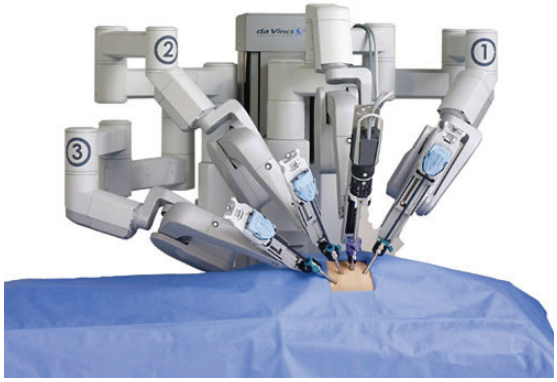
[9] `http://en.wikipedia.org/wiki/Flexible_manufacturing_system`

flexibility at the price of significant extra system cost and complexity.

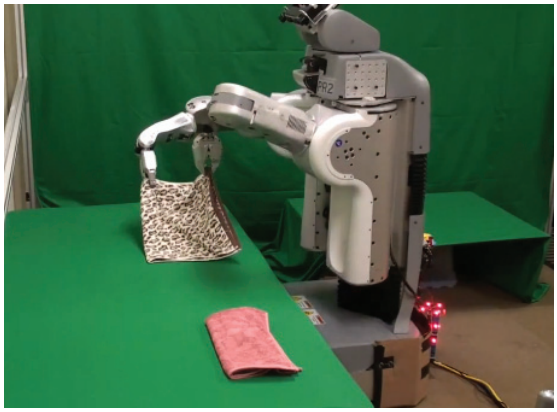


(a)

(b)



(c)



(d)

Figure 1.2: Anthropomorphic and functional approaches to human-like robots. (a) Honda Asimo robots playing a soccer game. (b) Ishiguru and his android copy (which one is real?). (c) The da Vinci surgical system. (d) The PR-2 robot from Willow Garage, with algorithms and arms for cloth folding.

At the other end of the spectrum, instead of skirting around uncertainty, practitioners in robotics, influenced by research in artificial intelligence, also seek to build general purpose robots that are highly reconfigurable, much like today's personal computers which can handle a broad array of computing tasks. There seems to be two general trends in this endeavor. The first one is the anthropomorphic approach: Study how human do things and replicate the same mechanism and design on robotic counterparts (see Fig. 1.2(a),(b) for some examples).

At present, success in this approach is limited; the produced robots are mostly human-like by look, not function. There is a good reason for this. The human anatomy is but one of many possible designs at the task of adapting to the woes of nature; it is not obvious that this design should be the most efficient at all tasks. This brings us to the alternative route which focuses directly on the task (or function) to be performed and designs around just the more important aspects. Remarkable achievements have been accomplished following this paradigm, partially validating the feasibility of this approach. We now have robots that can carry out delicate tasks under high levels of uncertainty. The da Vinci surgical system (see Fig. 1.2(c)) enables surgeons to perform operations at precisions that are previously impossible. The PR2 modular robot platform (see Fig. 1.2(d)) allows highly complex tasks to be done, such as cloth folding [17]. These are just two examples from a continuously growing list of programmable (thus flexible) machines capable of high precision and sophisticated operations.

What makes the da Vinci and the PR2 special? Taking a closer look, it is not hard to see that they are somewhat similar: Both platforms have the basic design with a base, vision system, and arms, reflecting the fact that mobile bases, good vision/sensing systems, and configurable arms are indispensable components of general purpose robots. What distinguishes them from high precision assembly lines is the addition of elaborate sensors and lightening fast computing units. At present, however, sensors are largely abused/underutilized in the sense that the majority of information collected by sensors may end up being ignored and/or discarded [18]. For example, to grasp an object, it appears that we only need to have a rough geometric model of the object; we hardly need to know its colors or the patterns on its surface. Therefore, for this grasping task, laser scanner data is much more precise and compact than that from a high definition camera. Yet frequently, multiple laser scanners and cameras are packed onto a system without understanding whether it is indeed necessary. Besides basic design issues, the overloading sensors also tax the system unnecessarily in many

other ways, including additional cost of sensors, a larger footprint, more computation, and increased energy consumption.

In this thesis, we argue that one key to overcoming the sensor design/modeling challenge is to obtain a thorough understanding of the inherent information required by the tasks to be performed. Our motivations behind this pursuit are threefold. First, unlike some other subfields of basic robotics research such as motion planning, our understanding of sensors and the information they produce is far from comprehensive. Studying this branch of robotics, besides satisfying intellectual curiosity, helps us gain a firmer grasp of sensing as one of the three essential building blocks (sensing, computation, and actuation) of robots. Second, adequate knowledge of a task's *information* requirement will help us design more economical robots by removing unnecessary sensing and computation elements early in the design. Third, as robots get more and more complex in response to the tasks we assign them, a hierarchical design is likely unavoidable. The understanding that we seek enables the separation of the information gathering (sensing) process from the decision making (planning) process, which will in turn enable a hierarchical and modular approach for building more capable robots.

1.2 Key Themes

The primary focus of this thesis is on ***combinatorial filters*** and more precisely, ***how to systematically analyze the combinatorial structure of sensor data induced by tasks and apply the structural insight to solve these tasks efficiently via combinatorial filtering, guided by a principled, information space-based methodology***. Unlike other approaches at sensor fusion, our main interest is not with building probabilistic models of sensors to maximize the expected signal-to-noise ratio. Rather, we start with a task and an abstract but realistic sensor model. Via carefully looking at the structure of the data *induced* by the given task, we design algorithms to effectively trim away irrelevant

information while retaining *all* relevant information for solving the given task. As such, our method does not suffer from issues such as getting trapped in a local stationary point. When it is possible, we also seek to maintain a minimal set of sufficient information from the data. Our minimalist philosophy manifests itself in both information spaces and combinatorial filters.

1.2.1 Information spaces

The information space notion, as explained in [3], was introduced in the context of sequential games (see [19, 20]), and has been used in various contexts, including stochastic control theory [21, 22] and pursuit-evasion games [23]. Information spaces are analogous to configuration spaces from classical motion planning. Let X denote a *state space* for a given system. The world that we define may contain any number of *sensors* that map states to observations. Typically, a *sensor mapping* is defined as $h : X \rightarrow Y$, in which Y is an *observation space* (the set of possible sensor readings). An *observation history*, $\tilde{y}$, is a parameterized family of observations, usually taken over time by the sensor. Typically, $\tilde{y}_t$ is defined as $\tilde{y}_t : [0, t] \rightarrow Y$, which yields *time-parameterized* observations from time 0 to time t. To actuate agents in the world, we define a *control space* or *action space* U. We may apply some *control history* $\tilde{u} : T \rightarrow U$ over an interval of time T. The effect on the state can be modeled as $\dot{x} = f(x, u)$ with the cumulative outcome $x = \Psi(x(0), \tilde{u})$.

Unfortunately, we must assume in our scenario that the state x is *not* available as input for decision making. Instead, at some time t, the pair $\tilde{y}_t$ and $\tilde{u}_t$, along with any *initial conditions*, constitute the complete information available for solving tasks. Hence, $\eta_t = (\tilde{y}_t, \tilde{u}_t)$ is called the *history information state*. Let the *history information space* $\mathcal{I}_{hist}$ be the set of all η_t for all allowable t. Any of these three sources of information—the initial condition, sensor observations, and control inputs—may give rise to uncertainty (nondeterministic or probabilistic) to various degrees: At one end, we may have no initial condition, faulty sensors,

and imprecise controllers. Completing a task can be quite challenging in this situation. At the other end, we may have precise initial state, full knowledge of the environment, and perfect control, in which case a feasible control strategy becomes trivial to obtain; even a globally optimal control strategy is possible.

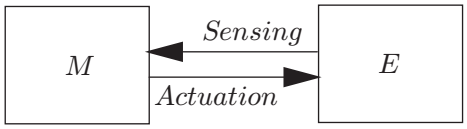


Figure 1.3: The state of the environment is not known. The only information available to make inferences is the history of sensor observations, actions that have been applied, and the initial conditions. This history becomes the *information state*.

Capturing the basic interactions between a system and its environment (see Fig. 1.3), the information space formulation is quite general. For example, one may choose to reconstruct x using η_t which corresponds to *state estimation* frequently assumed in feedback control. We adopt an alternative route via designing a smaller information space, say $\mathcal{I}_{der}$, by collapsing, manipulating, and encoding $\mathcal{I}_{hist}$ to allow a fixed task to be accomplished with the lowest complexity and greatest robustness possible, which avoids state estimation. Each reduced information state $\eta' \in \mathcal{I}_{der}$ may contain insufficient data to reconstruct x but must nevertheless be sufficient for the task.

1.2.2 Combinatorial filtering

In signal processing, a *filter* is defined as a device or process that removes from a signal unwanted feature. Bayesian filters such as the Kalman filter, the extend Kalman filter, and the unscented Kalman filter have seen wide applications in robotics [24, 25, 26, 27, 28, 10]. For example, the vehicles from the DARPA Grand Challenge race heavily employed point cloud estimation using algorithms such as Simultaneous Localization and Mapping (SLAM) [10].

Although Bayesian filters prove to be quite successful in practice, at times they also

tend to obscure the inherent structure of the information required to solve the given task. For example, even when the state transition and observation models are highly non-linear, applying the extended Kalman filter may still yield, statistically speaking, satisfactory solutions to state estimation problems. This has lead to abusive applications of Bayesian filters without carefully justifying whether the assumptions of the employed Bayesian filter are met. As such, these applications do not offer much insight in understanding what information is truly needed in completing a given task, which limits their contribution in helping making better robots.

Combinatorial filtering, we argue, forces a principled approach. By combinatorial filtering, we refer to the process of removing from sensor data a portion in such a way that, given what is left, the discarded data offers absolutely no additional information for solving the task at hand. In building such filters, the first step is always to find a more refined, derived information space, $\mathcal{I}_{der}$, as a subspace of the history information space $\mathcal{I}_{hist}$. After that, combinatorial filtering is simply processing the last information state in $\eta'_{t-1} \in \mathcal{I}_{der}$, the latest control u_{t-1}, and the latest observation y_t to obtain $\eta'_t \in \mathcal{I}_{der}$. We note that it is important that $\mathcal{I}_{der}$ is a subspace of $\mathcal{I}_{hist}$ (i.e. *complete*); this ensures that with η'_t, we can discard η_t and "live" in $\mathcal{I}_{der}$. Under this framework, solving a task becomes finding the correct abstract information space, applying the associated filter, and making control/actuation decisions based on the latest information state $\eta'_t \in \mathcal{I}_{der}$.[10]

$$
\begin{array}{ccccccc}
I_{hist} & \cdots \rightarrow & \eta_{t-1} & \overset{u_{t-1},y_t}{\longrightarrow} & \eta_t & \rightarrow \cdots \\
\downarrow & & \downarrow & & \downarrow & \\
I_{der} & \cdots \rightarrow & \eta'_{t-1} & \overset{u_{t-1},y_t}{\longrightarrow} & \eta'_t & \rightarrow \cdots \\
\downarrow & & \downarrow & & \downarrow & \\
\cdots & \cdots \rightarrow & \cdots & \longrightarrow & \cdots & \rightarrow \cdots
\end{array}
$$

Figure 1.4: Although it is possible to obtain $\eta'_t \in \mathcal{I}_{der}$ from $\eta_t \in \mathcal{I}_{hist}$, it is also possible to derive it from η'_{t-1} and $u_{t-1,t}, y_{t-1,t}$: The diagram commutes.

[10] A combinatorial filter, inspired by Bayesian filters, can be considered as a sibling of Bayesian filters such as a Kalman filter: In a Kalman filter, η' represents mean and covariance, rather than a full pdf. In our context, however, the information space may be much "smaller" and stochastic models are not necessary.

The above discussion on information space and combinatorial filtering is summarized in Fig. 1.4. At the top level, we always have $\mathcal{I}_{hist}$, the historical information space that by assumption has all relevant information. Then, depending on the given task, we may be able to discover one or more derived information spaces, $\mathcal{I}_{der}$, $\mathcal{I}_{der}{}'$, ..., such that each of the derived information spaces still captures the information necessary for solving the task. The arrows represent the combinatorial filtering process. With this diagram, it is straightforward to see the power of this framework: To solve the given task, we just need to pick a sensor that can provide enough information for constructing one of the derived information spaces.

1.3 Overview of the Thesis

In exploring combinatorial structures and filter design in information spaces, this thesis aims to investigate realistic abstractions of two practical task classes: (1) visibility based pursuit evasion games, and (2) sensor network based story validation (a form of path inference) and the inverse problem of optimal sensor placement.

1.3.1 Shadow information space: Combinatorial filters for tracking hidden targets

In [29], the authors made the following observation on the general interest in pursuit evasion games: *"Chases and escapes, or pursuits and evasions, are activities which are firm parts of our everyday or fantasy life. When young we are playing the game of tag or the game of hide-and-seek and watch Tom & Jerry cartoons. Pursuit-evasion problems remain fascinating for most of us even when we grow older, and it is not surprising that in half of the Hollywood movies good guys are chasing the bad guys, while in the remaining half bad guys are chasing the good ones."* Besides great interest in the public domain, pursuit evasion has also remained a topic of great academic interest. The mathematical study of pursuit games dates back to at least four decades, with its roots in differential games [30, 31, 32]. Subsequent studies have

focused on classical differential games [33, 34, 35, 36], pursuit evasion on graphs [37, 38, 39], visibility based pursuit evasion games [40, 41, 23, 42, 43, 44, 45, 46, 47, 48], and various other formulations [49, 50, 51, 52, 53, 54].

Directing our attention to visibility based pursuit evasion, it contains two key interacting ingredients: Passively estimating the distribution of hidden targets and actively planning to reduce the uncertainty of this distribution. The general goal in research on visibility based pursuit evasion is to algorithmically clear evading targets from a workspace. As pursuers try to ensure that the workspace is evader-free, they always need to maintain the pursuit status, remembering whether a region outside the pursuers' field-of-view is contaminated (may contain evaders) or clear (does not contain evaders).

The first half of this thesis is devoted to the passive ingredient of pursuit evasion games, studying how the distribution of hidden targets residing in unobservable regions of the environment evolves. In particular, we introduce the notion of *filters* over *shadow information spaces* for tracking moving targets in unobservable regions, as a generalization of this aspect of visibility based pursuit evasion. To approach our goal, information contingent to task completion is first extracted from sensor observation history and compressed in a lossless fashion for storage and effective computation. Next, depending on whether the targets of interest are moving nondeterministically or probabilistically, concrete problems are formulated and solved by carefully manipulating and fusing observation and data. At a higher level, at any time, our algorithm can estimate the number of targets hidden in regions that are not directly observable.

Connecting to the main themes of this thesis, two derived information spaces, the *shadow sequence information space* ($\mathcal{I}_{ss}$) and the *bipartite information space* ($\mathcal{I}_{bip}$) are constructed, both of which contain a much more compact description of the necessary information for tracking the distribution of hidden targets, with $\mathcal{I}_{bip}$ a subspace of $\mathcal{I}_{ss}$. After these derived information spaces are discovered, we build filters that allow us to "live" in these spaces.

Depicting it in pictures similar to that in Fig. 1.4, the hierarchy is illustrated in Fig. 1.5.

$$
\begin{array}{lllll}
I_{hist} & \cdots \rightarrow & \eta_{t-1} & \overset{u_{t-1},y_t}{\longrightarrow} & \eta_t \quad \rightarrow \cdots \\
\downarrow & & \downarrow & & \downarrow \\
I_{ss} & \cdots \rightarrow & \eta'_{t-1} & \overset{u_{t-1},y_t}{\longrightarrow} & \eta'_t \quad \rightarrow \cdots \\
\downarrow & & \downarrow & & \downarrow \\
I_{bip} & \cdots \rightarrow & \eta''_{t-1} & \overset{u_{t-1},y_t}{\longrightarrow} & \eta''_t \quad \rightarrow \cdots \\
\downarrow & & \downarrow & & \downarrow \\
\cdots & \cdots \rightarrow & \cdots & \longrightarrow & \cdots \quad \rightarrow \cdots
\end{array}
$$

Figure 1.5: The shadow information space structure.

1.3.2 Sensor network based verification and path inference

In computer science, robotics, and control, a frequently encountered problem is verifying that an autonomous system, be it a program or a robot, is performing as designed. For example, a service robot may plan a path to clean office rooms one by one. Due to internal (sensor/actuator/computing units malfunctioning) or external factors (strong electromagnetic interference, for example), the robot may mistake one room for another and fail to accomplish its task without knowing that it has failed. A robot or a system may also be compromised for malicious purposes, producing intentionally bogus records of its actual path to hide the fact. In such cases, it would be highly desirable if external monitoring could automatically determine that a robot has faltered.

In the second half of this thesis, we introduce realistic abstractions of the aforementioned problems (see Fig. 1.6 for an example) and show that such formulations are computationally tractable. Specifically, one or more agents (robots or people) are assumed to move in an indoor environment, of which regions are monitored by external sensors (beam detectors and occupancy sensors). We assume that the agents are not aware of these sensors. From a story told by an agent, which is a sequence of places in the environment it has visited, and combined recordings of these sensors, we provide polynomial time algorithms (with respect to the complexity of the environment, the length of the story, as well as the length of the

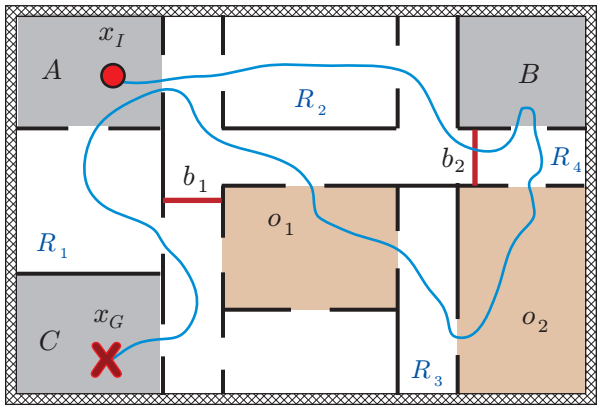


Figure 1.6: A workspace with two beam detectors b_1, b_2, two occupancy sensors o_1, o_2, and three labeled rooms A, B and C. An agent's story can take the following form, for example: "I started from room A, went to room B, then room A and finally arrived room C." This story would be consistent with the blue path from x_I to x_G, which triggers sensor recordings b_2, o_2(activation), o_2(deactivation), o_1(activation), o_1(deactivation). There are four connected components R_1 through R_4 when regions guarded by sensor range and rooms in agent x's story are treated as workspace obstacles.

observation history) for the inference problem of whether the given story is consistent with the sensor recordings.

If one assumes that the behavior of a set of moving bodies is largely unknown, the proposed problem becomes inferring various properties of these moving bodies with a network of simple sensors. Binary proximity sensors have been employed to estimate positions and velocities of a moving body using particle filters [55] and moving averages [56]. The performance limits of a binary proximity sensor network in tracking a single target are discussed and approached in [57], followed by an extension to the tracking of multiple targets [58]. The task of counting multiple targets is also studied under different assumptions [59, 60]. In these works, the sensor network's aggregate sensing range must cover the targets of interest at all times, which is much more difficult to implement than guarding critical regions of an environment. When only subsets of an environment are guarded, *word problems in groups* [61, 62] naturally arise. For the setup in which targets moving inside a 2D region are monitored with a set of detection beams, [63] characterizes possible target locations,

target path reconstruction up to homotopy, and path winding numbers. In this domain, the surfacing of more interesting behaviors also induces an increase in complexity; few efficient algorithms exist. This prompts us to ponder: Can we do better if partial knowledge of a target's behavior is available?

On the other hand, if sensors external to moving bodies are ignored, one is left with the task of systematically verifying that the moving bodies do not have unexpected behaviors. Complex moving bodies such as robots are often modeled as hybrid systems. Existing verification techniques either address subclasses of hybrid systems or approximate reachable sets of such systems [64, 65, 66], because the problem of verifying a system with continuous state space and control input is generally undecidable [67]. In practice, this difficulty translates into the necessity of external measures to safeguard the unverified portion of a system. Alternatively, when high level task specifications can be coded as General Reactivity(1) formulas [68], the task of composing controllers into verifiably correct hybrid automata can be carried out automatically using linear temporal logic [69, 70]. Even for such provably correct designs, malfunction can still occur due to sensor/actuator/computer errors. Keeping these systems in check again requires monitoring with external sensors.

In exploring the sensor based story validation and path inference problems, we again find ourselves designing combinatorial filters based on the most efficient representation of the information necessary for completing the given task(s). In this portion of the thesis, more emphasis is put on filter design. As we will observe in these problems, there are two distinct data structures coming from the formulations (between which there are no set inclusion relationship). Although it is possible to build filters with each of these structures, more performance is managed from a mixed filter that fully exploits the structural property of the problems. Finally, we also explore the inverse problem of optimal sensor placement for path inference and prediction.

CHAPTER 2

SHADOW INFORMATION SPACES

2.1 The Problem of Tracking Hidden, Moving Targets

In a game of *hide-and-seek*, after the hiders conceal themselves, the seekers, familiar with the environment, start their search for the hiders. Most who played the game as schoolchildren know that an effective search begins with the seekers checking places having high probabilities of containing hiders, from previous experience: A closet, an attic, a thick bush, and so on. After the most likely locations are exhausted, the next step is to carry out a systematic search of the environment, possibly with some seekers guarding certain escape routes. Occasionally, during the game, hiders may attempt to relocate themselves to avoid being found. Although the hiders succeed sometimes, they could end up being spotted by the seekers and are instead getting found earlier.

Although a child's play, the game of hide-and-seek captures the two key interacting ingredients of pursuit evasion games: Passively maintaining/estimating the distribution of hidden targets and actively planning to reduce the uncertainty of this distribution. The general goal in pursuit evasion research is to algorithmically clear evading targets from a workspace. As pursuers try to ensure that the workspace is evader-free, they always need to maintain the pursuit status, remembering whether a region outside of the pursuers' field-of-view (FOV) is contaminated (may contain evaders) or cleared (does not contain evaders), which can be coded with one bit of information per region.

Chapters 2 and 3 focus exactly on this passive ingredient of pursuit evasion games,

reasoning about the information residing in unobservable regions of the environment. In particular, we introduce the notion of *filters* over *shadow information spaces* for tracking moving targets in unobservable regions, as a generalization of this aspect of pursuit evasion. To achieve this, we first process sensor observation history and compress it in a lossless fashion for our task classes, for storage and effective computation. Next, depending on whether the targets of interest are moving nondeterministically or probabilistically, concrete problems are formulated and solved by carefully manipulating and fusing observation and data. At a higher level, at any time, our algorithm can estimate the number of targets hidden in regions not directly observable. We note that, although the active problem of planning a pursuit path is not addressed in this work, heuristic search strategies can be readily implemented on the space of filter outputs. The work in these two chapters is based on [71, 72, 73].

2.1.1 Pursuit evasion games

The mathematical study of pursuit evasion games dates back at least four decades, with its roots in differential games [30, 31, 32]. Although optimal strategies for differential pursuit evasion games are still actively pursued [33, 34, 35, 36], a variant of differential pursuit evasion games, visibility based pursuit evasion, has received much attention recently. Development of visibility based pursuit evasion games can be traced back to [37], in which a pursuit evasion game on a discrete graph is introduced with the goal of sweeping evaders residing on continuous edges of the graph. The evaders can move arbitrarily fast, but must move continuously. The watchman route problem [38, 39], formulated twelve years later as a variant of the art gallery problems [74, 75], involves finding shortest route to clear static intruders. An intruder is considered cleared if a line of sight exists between the intruder and a point of the watchman route.

Influenced by these two threads of research, [40] defined what we know today as visibility

based pursuit evasion games in which the discrete graph domain is replaced by a path connected interior of a 2D polygon and a continuously moving evader is considered to be cleared if it falls into the visible region of a pursuer (in this case the pursuer is equipped with two flashlights and is called a 2-searcher). Thinking along the same lines as the art gallery problems, it was soon established that for a pursuer with an omni directional infinite range sensor (an ∞-searcher), it is NP-hard to decide the minimum number of pursuers needed for the class of multiply connected polygons [41, 23]. The insight that bitangents and inflections fully capture the critical changes leads to a generalization from polygons to curved environments [42] and form the basis of some critical events used in our work.

Various sensing and motion capabilities are explored in visibility based pursuit evasion. Interestingly, it turns out that a 2-searcher is as capable as an ∞-searcher in simple polygons [43]. Pursuers with a single flashlight (a 1-searcher) are investigated in detail in [44] and [45], with the latter limiting the pursuer's motion to the boundary of the environment. Variations along this line include limited FOV [46], unknown environments [47], and bounded speed [48]. Another theme in pursuit evasion games is to discretize time and put speed bounds on both pursuers and evaders. In this setting, sufficient conditions and strategies for a single pursuer to capture an evader are given to the classical lion-and-man problem in the first quadrant of the open plane [49]. This problem is then extended to $\mathbb{R}^n$ and multiple pursuers in [50], and multiple pursuers with limited range in [51]. In [76], a multi-vehicle rendezvous problem is studied with the vehicles having fixed field-of-view and limited non-holonomic motion model. Finally, pursuit evasion is also studied in the probabilistic context [52, 53] and abstract metric spaces [54].

Since we provide algorithms for tracking moving targets, our work is also closely related to target tracking and enumeration. The problem of accurately counting the number of targets with overlapping footprints is solved with a novel approach of integrating over Euler characteristics in [77]. With a virtual sensor that reports visible features of polygonal envi-

ronment as well as indistinguishable targets, static targets are counted under various setups in [78]. A filtering algorithm is provided in [58] to count moving targets with a network of binary proximity sensors. In [79], Simultaneous Localization and Mapping (SLAM) and Detection and Tracking of Moving Objects (DTMO) are combined to attack both problems at the same time. A specialization of our problem is investigated in [63] in which the sensor FOV becomes one-dimensional beams. Real-time people counting with a network of image sensors is studied in [80].

Another research area of relevance to this chapter, especially the probabilistic formulations we give in Section 3.2, is the study of *optimal search* [81], which proposes a Bayesian approach for maintaining a target distribution and use that information for guiding the planning of optimal search paths. The essential idea from optimal search is to plan a path to eliminate regions with highest probability of containing the targets. In doing so, optimal search algorithms allow the prediction of the *no-detection likelihood* [82, 83, 84, 85], which is the probability that the targets remain undiscovered at given stages of a search effort, even before the actual search is carried out. Although our work also seeks to maintain a target distribution along a given path, we focus on the computational problem of how *topological* changes of non-observable components, which are *combinatorial* in nature, can be correctly and efficiently processed as the target distribution evolves. This topological and combinatorial element of target tracking exists whether the problem formulation is probabilistic or not. In this aspect, the problems we address here are mostly orthogonal to classical optimal search problems, which cover environments (support surfaces of the distribution) that are mainly two-dimensional, obstacle-free planes such as these appearing in typical maritime applications. As such, the results presented in this chapter should benefit the extension of optimal search results to covering more diverse workspaces such as urban areas and hilly terrains, where topological changes of non-observable components are frequent.

2.1.2 Our contribution

Our motivation in this work comes from the observation that the passive task of maintaining target distribution in visibility based pursuit evasion games has not received the full attention it deserves. Although an integrated approach, taken by prior research, may offer better performance on individual problems, the structures of the underlying information space that are similar in many of these problems has not been fully explored and made reusable.

The main contributions of this work are twofold. First, as explained previously, we generalize visibility based pursuit evasion by introducing a richer class of problems and providing a framework as a submodule for systematically attacking these problems. Second, the capability of effectively tracking hidden, moving targets, a general type of *situation awareness*, applies to a large class of time-critical tasks in both civilian and national security applications. For example, in a fire evacuation scenario, knowing the possible or expected number of people trapped in various parts of a building, firefighters can better decide which part of the building should be given priority when they coordinate the search and rescue effort.

The rest of the chapter is organized as follows. Section 2.2 provides a mathematical definition of what we mean by "shadows" and "component events," which can be best captured using a chronological sequence. Section 2.3 suggests the general problem of tracking hidden targets after bringing in moving targets and FOV events. Section 2.4 formulates the problem of estimating the number of targets hidden in shadows for nondeterministically moving targets and establishes its polynomial time solvability using results from integer linear programming theory. With these necessary ingredients, Section 2.5 formally describes the full *shadow information space* structure.

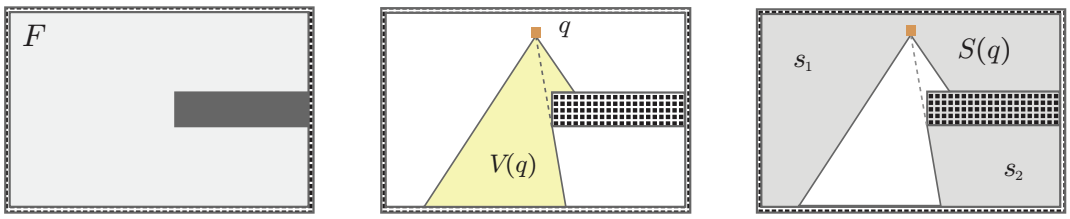
2.2 Component Events: The Space-Time Perspective

Intuitively, in the hide-and-seek game, the part of the world that is not observable is comprised of many components, each of which has a life span. To study the information flow in them, a formal definition of components is first in order; the temporal relationship among them then naturally comes up.

2.2.1 Visibility event induced shadow components

Let a nonempty set of robots move along continuous trajectories in a *workspace*, $W = \mathbb{R}^2$ or $W = \mathbb{R}^3$. Let the configuration space of the robots be $\mathcal{C}$. At some time t, there may be configuration space obstacles $\mathcal{C}_{obs}$ which may vary over time, leaving $\mathcal{C}_{free} := \mathcal{C} \backslash \mathcal{C}_{obs}$ as the free configuration space. Let $q \in \mathcal{C}_{free}$ be the configuration of the robots at time t. Returning to the workspace, there is a closed *obstacle region* $O(t) \subset W$, leaving $F(t) := W \backslash O(t)$ as the free space. The robots are equipped with sensors that allow them to make shared observations in a joint *FOV* or *visible region* $V(q,t) \subset F(t)$. For convenience, we take the closure of $V(q,t)$ and assume that the visible region is always closed. Let $S(q,t) := F(t) \backslash V(q,t)$ be the *shadow region*, which may contain zero or more nonempty path connected components (path components for short). A path component is assumed to be nonempty unless otherwise specified. At any instant, $O(t), V(q,t)$, and $S(q,t)$ have disjoint interiors by definition and $W = O(t) \cup V(q,t) \cup S(q,t)$. Figure 2.1 shows $V(q), S(q)$ for a point robot holding a flashlight with $F \subset W = \mathbb{R}^2$, $\mathcal{C}_{free} \subset SE(2)$, which is the set of two-dimensional translations and rotations (here we omit the parameter t from F, V, and S since the obstacle region does not vary over time).

To observe how path components of the shadow region evolve over time, let the robots follow some path $\tau : [t_0, t_f] \to \mathcal{C}$ in which $[t_0, t_f] \subset T \subset \mathbb{R}$ is a time interval. Let $Z := W \times T$ denote the workspace-time space. We may let $O : T \to Pow(Z)$ be the map that yields the



(a) (b) (c)

Figure 2.1: (a) The environment and the free space, F. Note that in this example, the obstacle region is fixed; therefore, F is constant. (b) The visible region, $V(q)$. (c) The shadow region, $S(q)$, with two path components s_1, s_2.

obstacle region and define V, S as

$$V, S : \mathcal{C} \times T \to Pow(Z).$$

Since a path τ, parameterized over $t \in T$, is always assumed, with slight abuse of notations, we write $V(t), S(t)$ in place of $V(\tau(t), t), S(\tau(t), t)$, respectively. In particular, we are interested in $S(t)$ and call it a (time) *slice*. For any $(t_a, t_b) \subset [t_0, t_f]$, let the union of all slices over the interval,

$$S(t_a, t_b) := \bigcup_{t \in (t_a, t_b)} S(t),$$

be called a *slab*, which is an open subset of Z. For any subset z of Z, define its projection onto the time axis as

$$\pi_t : Pow(Z) \to Pow(T),$$
$$z \mapsto \{t \mid (p, t) \in z \text{ for some } p \in W\}.$$

Let $s_{t,i} \subset S(t)$ denote the i-th path component of $S(t)$ (assuming some arbitrary ordering). Let $s_{i'}$ denote the i'-th path component of a slab $S(t_a, t_b)$ (again, assuming some arbitrary ordering). $S(t_a, t_b)$ is *homogeneous* if for all $t \in (t_a, t_b)$ and all i, there exists i' such that

$$s_{t,i} = S(t) \cap s_{i'},$$

and separately,

$$\pi_t(s_{i'}) = (t_a, t_b) \text{ for all } i'.$$

A homogeneous slab is called *maximal* if it is not a proper subset of another homogeneous slab. The definition then partitions $S(t_0, t_f)$ into some m disjoint maximally homogeneous slabs plus some slices

$$S(t_0, t_f) = S(t_0, t_1) \cup S(t_1) \cup S(t_1, t_2) \cup \ldots \cup S(t_{m-1}) \cup S(t_{m-1}, t_f).$$

That is, homogeneity of $S(t_0, t_f)$ is broken at $t_1, \ldots, t_{m-1}$. What exactly happens at t_1, …, t_{m-1}? Let there be two homogeneous slabs $S(t_a, t_b)$ and $S(t_b, t_c)$ such that for some $k \in \{1, \ldots, m-1\}$, $t_{k-1} \leq t_a < t_b = t_k < t_c \leq t_{k+1}$. Let s_i be an arbitrary path component of $S(t_a, t_b)$, at $t = t_b$, s_i may: (1) *live on*, if there exists a path component $s_j \subset S(t_b, t_c)$ such that $\overline{s_i} \cap S(t_b) = \overline{s_j} \cap S(t_b) \neq \varnothing$ ($\overline{s}$ denotes the closure of s), or (2) *disappear*, if $\overline{s_i} \cap S(t_b) = \varnothing$. Similarly, a path component $s_j \subset S(t_b, t_c)$ may *appear*, if $\overline{s_j} \cap S(t_b) = \varnothing$. Finally, a nonempty set of path components $\{s_i\}$ of $S(t_a, t_b)$ may *evolve*, if there is a nonempty set of path components $\{s_j\}$ of $S(t_b, t_c)$ such that $|\{s_i\}| + |\{s_j\}| \geq 3$ and $\bigcup_i \overline{s_i} \cap S(t_b) = \bigcup_j \overline{s_j} \cap S(t_b) \neq \varnothing$ is a single path component of $S(t_b)$.

By definition, appear, disappear, and evolve are critical changes that only (and at least one of which must) happen between two adjacent maximally homogeneous slabs. We call these changes *component events*. With component events, homogeneity and maximality readily extend to path components of slabs. A path component $s_i \subset S(t_a, t_b)$ is called *homogeneous* if no component events happen to a subset of s_i in (t_a, t_b); s_i is called *maximal* if it is not a proper subset of another homogeneous path component.

At this point, a type of general position is assumed to avoid two tedious cases: (1) four or more path components cannot be involved in an evolve event, and (2) two or more component events cannot occur at the same time. In practice, non-general position scenarios form a

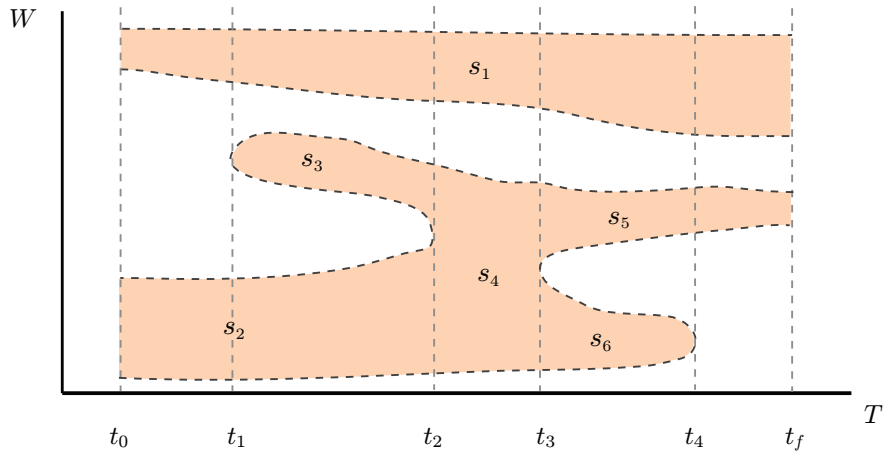


Figure 2.2: Evolution of shadows. At t_1, s_3 appears. At t_2, s_2 and s_3 merge into s_4. At t_3, s_4 splits into s_5 and s_6. At t_4, s_6 disappears. The "sizes" of the shadows have no effect on the critical events.

measure zero set and can be dealt with via small perturbations to the input if required. With such an assumption, exactly one component event happens between two maximally homogeneous slabs. Moreover, the evolve event can be divided into two sub-events: *split* if $|\{s_i\}| = 1, |\{s_j\}| = 2$ and *merge* if $|\{s_i\}| = 2, |\{s_j\}| = 1$.

We now piece together the definitions using an example illustrated in Fig. 2.2. Restricting to the time interval (t_0, t_f), there are five maximally homogeneous slabs, $S(t_0, t_1)$, ..., $S(t_4, t_f)$. Certain proper subsets of one of these, such as $S(t_2^+, t_3^-)$ with $t_2 < t_2^+ < t_3^- < t_3$, are again homogeneous but no longer maximal; on the other hand, supersets such as $S(t_2^-, t_3^+)$ with $t_2^- < t_2 < t_3 < t_3^+$, are no longer homogeneous. The slab $S(t_0, t_f) \subset Z$ has two path components (s_1 and $\mathrm{Int}(\overline{s_2} \cup \overline{s_3} \cup \overline{s_4} \cup \overline{s_5} \cup \overline{s_6})$, in which Int denotes the interior of a set) and six maximally homogeneous path components $s_1, \ldots, s_6$. The intersection of a vertical line at $t \in (t_0, t_f)$ with $S(t_0, t_f)$ corresponds to the slice $S(t)$. For convenience, it is assumed that $t = t_0$ is not a critical time in the sense that for each path component $s_{t_0,i} \subset S(t_0)$, $s_{t_0,i} = \overline{s_{i'}} \cap S(t_0)$ for some path component $s_{i'} \subset S(t_0, t_1)$. A similar assumption is made for $t = t_f$. Under this setup, there are four component events: (1) s_3 appears at $t = t_1$, (2) s_2 and s_3 merge to form s_4 at $t = t_2$, (3) s_4 splits into s_5, s_6 at $t = t_3$, and (4) s_6 disappears at

$t = t_4$. In contrast, the path component $s_1 \cap S(t_0, t_1)$ lives on through $t_1, \ldots, t_4$.

Finally in this subsection, we define the main concept of the chapter: *shadow*. It is easy to see that maximally homogeneous path components are pairwise disjoint. Let such a path component be called a *shadow* and let $\{s_i\}$ be the set of shadows of $S(t_0, t_f)$; note that $S(t_0, t_f)$ is contained in the closure of $\cup_i s_i$. In the preceding example, $\{s_i\} = \{s_1, \ldots, s_6\}$. For some $t \in (t_0, t_f)$, let a path component of $S(t)$ be labeled as $s_{t,i}$ if it is a slice of a shadow s_i. More precisely, $s_{t,i} = s_i \cap \{(p, t) \mid p \in W\}$. For $t = t_0$, $s_{t_0,i}$ is labeled such that $s_{t_0,i} = \overline{s_i} \cap S(t_0)$ for some path component $s_i \subset S(t_0, t_1)$. The same applies to the labeling of $s_{t_f,i}$. A path component of $S(t)$ has no label exactly when it is the border of two or more shadows of a slab. Since such labeling is unique, we drop time subscript of $s_{t,i}$ if t is fixed. In the rest of the chapter, we use the set $\{s_i\}$ to denote both shadows and slices of shadows; we simply call both types of path components *shadows* when no confusion arises from the context. When we do need to distinguish, the former will be called *workspace-time shadows* and the later *workspace shadows*.

2.2.2 Shadows are everywhere

To promote the intuition behind the mathematical definitions, let us look at a realistic example shown in Fig. 2.3(a). With the intention of guarding a planar region, spotlights are cast on the ground, creating a set of illuminated discs as shown. Assume that only the darker (orange) colored disc of light moves and follows the dashed line. For any position of the moving spotlight, the combined, illuminated set can be thought of as the FOV. Its complement in the plane is the shadow region, in which targets cannot be directly observed. Initially, there are two connected components, labeled s_1 (unbounded) and s_2, in the shadow region. As the spotlight moves along the dashed line, we observe that shadows may appear, disappear, merge, and split, as illustrated in Fig. 2.3(b) to (f). We constructed this example so that the events and evolution of shadows match exactly these of the example from Fig.

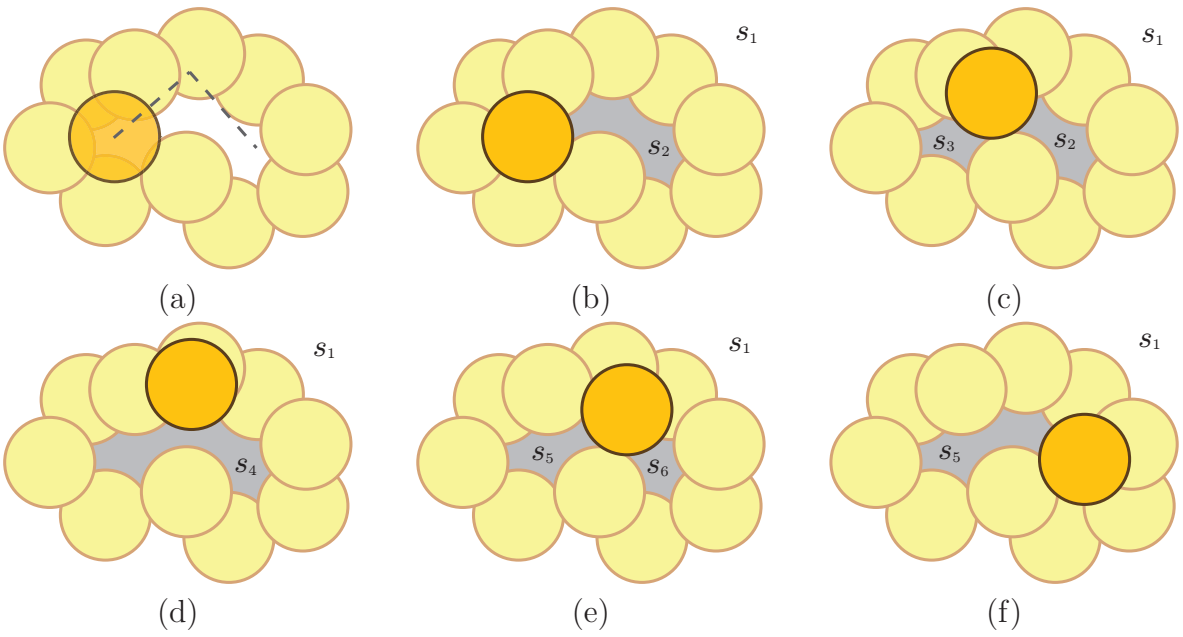


Figure 2.3: An example of shadows and their indexing/labeling. (a) The set of spotlights and the path to be followed by the darker (orange) colored spotlight. (b) Initially, only shadow s_2 and unbounded shadow s_1 exist. (c) A new shadow s_3 *appears*. (d) s_2, s_3 *merge* into a single shadow s_4. (e) s_4 *splits* into new shadows s_5, s_6. (f) s_6 *disappears*.

2.2.

The naive example suggests that shadows and component events arise from very simple setups. Indeed, shadows and component events are ubiquitous, showing up whenever moving sensors are placed inside environments. We provide three additional examples to corroborate this point; many others could be presented. In Fig. 2.4(a), omni-directional, infinite range sensors partition the 2D environment into polygonal shadows. The component events happen exactly when the sensors make *inflection* and *bitangent* crossings (see aspect graphs [86]), which gives rises to the concept of *gaps* and *gap navigation trees* as discussed in [87]. If the sensors have limited viewing angle [46] or limited range (Fig. 2.4(b)), alternate models governing visible and shadow regions are obtained. In Fig. 2.4(c), fixed infrared beams and surveillance cameras are placed inside a building, creating a set of three fixed shadows s_1, s_2, s_3. Such a setting is common in offices, museums, and shopping malls. As a last example, Figure 2.4(d) shows a simplified mobile sensor network with coverage holes. In this

case, the joint sensing range of the sensor nodes is the FOV and the coverage holes are the shadows, which fluctuate continuously even if the sensor nodes remain stationary (consider cellphone signals).

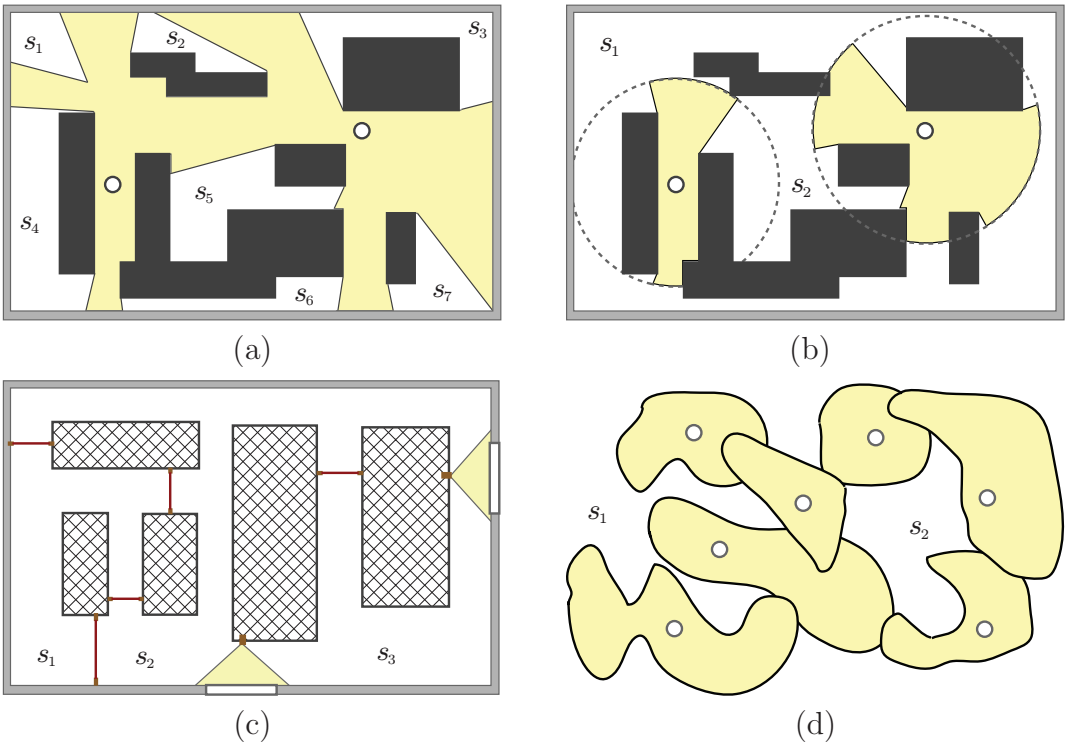


Figure 2.4: Real-world examples of shadow generating processes. (a) Two robots (white discs) carrying omni-directional, infinite range sensors. The free space is partitioned into seven shadows. (b) When sensing range is limited, the topology of shadows changes; only two shadows are left. (c) An indoor environment guarded by fixed beam sensors (red line segments) and cameras (yellow cones). There are three connected shadows. (d) A simple mobile sensor network in which the white discs are mobile sensing nodes, with shaded regions being their sensing range at the moment. There are two shadows with s_1 being unbounded.

For some environments, shadows are readily available or can be effectively computed with high accuracy, such as visibility sensors placed in 2D polygonal environments. In some other cases, shadows are not always easy to extract. As one example, estimating coverage holes in a wireless sensor network is rather hard since it is virtually impossible to know whether a point p is covered unless a probe is dispatched to p to check. It is also well known that 3D visibility structure is difficult to compute [88, 89]. Even though we do not claim to overcome

such inherent difficulties in acquiring visibility region and/or shadows, the method presented here applies as long as a reasonably accurate characterization of the shadows is available.

2.2.3 Shadow sequences as an abstract data structure

We conclude this section with the introduction of a *shadow sequence*, of which the importance will become more apparent in coming sections. When the shadows of $S(t_0, t_f)$ for a fixed path τ are put together, a sequential structure comes up. This structure, which we call a shadow sequence, captures the combinatorial changes of the labeled shadows through component events. A graphical illustration of the shadow sequence for the example in Fig. 2.3 is given in Fig. 2.5.

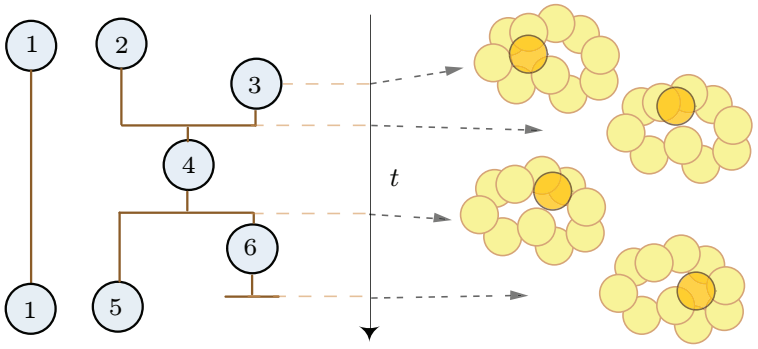


Figure 2.5: A graphical illustration of the shadow sequence for the example from Fig. 2.3. The numbers in the circles represent the labels of the shadows. The four events marked on the time line, from top to bottom, are appear, merge, split, and disappear. As one expects, this figure closely resembles Fig. 2.2.

2.3 Field-of-View (FOV) Events

2.3.1 Moving (and possibly hidden) targets

Our interest in shadows lies with maintaining information that is not directly observable by sensors. To effectively investigate how to track such information, we briefly characterize

what we mean by information. We assume that there is a non-negative integer number of *targets* in F, which are point entities that move arbitrarily fast but follow some continuous, unpredictable trajectories. The robots' sensors can detect certain *attributes* of these targets. We are interested in two types of attributes - location and identity.

Location. When the targets move in or out of the sensors' FOV, their appearance and disappearance may be detected. Depending on the sensors' capabilities, at least two levels of precision are possible:

1. The sensors can tell whether the FOV contains no target or at least one. In other words, each sensor's output is *binary* (motion detector is a sensor of this type).

2. Each target inside the FOV can be precisely located and counted.

Identity. When multiple targets are present, it may be possible to tell them apart. That is, the sensors may be able to *distinguish* the targets in the FOV. Roughly speaking, the targets may be

1. *Fully distinguishable.* When targets possess unique IDs recognizable by the sensors, they are fully distinguishable.

2. *Indistinguishable.* Although it appears that full distinguishability is the most powerful, it is not always available due to sensor cost constraint or even desirable due to concerns such as privacy. It is not hard to make targets indistinguishable: In the sensor output, erase any attributes that can be used to distinguish among the targets.

3. *Partially distinguishable.* Everything between the previous two notions of distinguishability belongs to this class. For instance, targets may form *teams* that are distinguishable by color.

Location and identity are related – full distinguishability implies that the sensors should be able to locate targets in the FOV. On the other hand, tracking locations over time can be

used to distinguish targets. However, these two attributes are not identical and it benefits to treat them orthogonally. For example, when colored teams of targets are present, a low resolution overhead camera can easily tell whether a team is present in the FOV via a color scan, acting as a combination of binary location sensor and identity sensor. Given sensors that can detect some subsets of the above mentioned attributes of targets, each labeled shadow can be assigned one or more variables that describe these attributes of the targets residing in the shadow. Note that although we deal mostly with binary and integer variables in this chapter, variables of other forms, such as real numbers, can also be incorporated over the structure of shadows and component events introduced here. When we consider targets in the shadows, a type of invariance arises:

Observation 1 *In an environment with only component events, the number of targets hidden in a workspace-time shadow is invariant over time; furthermore, a workspace-time shadow is a maximal set in which such invariance holds.*

By the assumption that a hidden target moves continuously, its trajectory is contained in the same workspace-time shadow when no component events happen. Two workspace shadows, as different time slices of the same workspace-time shadow, must intersect the same number of such trajectories since no target enters or exits the component in the time between the two timestamps associated with the two slices. This yields the invariance. The second claim follows the definition of workspace-time shadow as a maximal union of all such workspace shadows.

2.3.2 FOV events

If a location sensor also has *memory*, it will be able to detect changes to the number of targets in the FOV during a short time interval. We call such a change a *field-of-view event* (FOV event for short), which is a second type of critical events of our interest. Furthermore,

if the sensors know where an FOV event happens, these events can be associated with corresponding shadows. For a shadow s_i, three FOV events are possible: (1) a target *enters* s_i from the visible region, (2) a target *exits* s_i into the visible region, and (3) nothing happens at the boundaries between s_i and the visible regions (for a period of time), or *null* event.

Denoting the three FOV events as e_e, e_x, e_n, respectively, the collection of possible FOV events for a shadow s_i is the set

$$E_{FOV} = \{e_e, e_x, e_n\}.$$

Some sensors may only detect the enter and exit events explicitly, such as a sensing node in a sensor network that only senses targets passing through the boundary of its sensing range. For detection beams, the FOV is a line segment, which causes two FOV events to happen consecutively (see Figure 2.6). Certain systems may not have FOV events at all; an instance

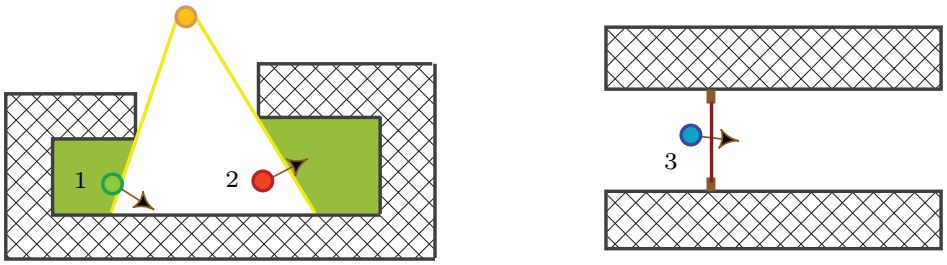


Figure 2.6: Illustration of FOV events for an environment with obstructed visibility (left) and for an environment with detection beams (right). (1) A target is about to exit a shadow into the FOV of the sensor (yellow disc). (2) A target is about to enter a shadow from the sensor's FOV. (3) A target is about to enter and exit the FOV of a beam sensor.

is a pursuit evasion game in which the evader always avoids appearing in the pursuer's FOV. The game ends when an evader is found or when it is confirmed that no evader is in the environment.

Since component events and FOV events both happen as robots move along some path τ in the free space F, it makes sense to treat them as a whole. It does not take much to represent them together: We can simply augment the shadow sequence to include the FOV

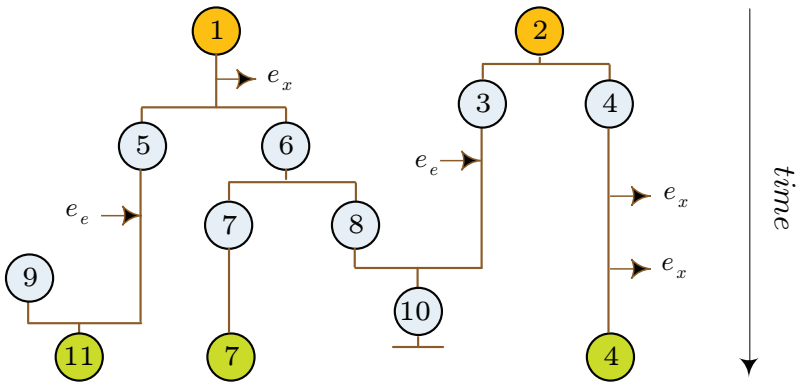


Figure 2.7: A typical sequence of critical events. The circles with numbers represent the shadows; the labeled arrows associate FOV events to shadows.

events. A typical combined sequence of critical events is shown in Fig. 2.7. To incorporate FOV events, the invariance from Observation 1 needs to be updated.

Observation 2 *In an environment with component and FOV events, the number of targets hidden in a workspace-time shadow is invariant between FOV events (excluding null events) associated with the shadow; the time span of such invariance is again maximal.*

To see why Observation 2 is true, note that an enter FOV event can be viewed as an appear component event immediately followed by a merge component event. Same breakdown holds for exit FOV events. The case is then reduced to Observation 1. Observation 2 establishes that for the task of tracking hidden targets that move continuously, any sensor data unrelated to critical events can be safety discarded without adverse effects.

2.3.3 The general problem formulation

With the introduction of component and FOV events, we can formally define the general problem of tracking hidden targets that move continuously. The following inputs are assumed:

1. An initial distribution of targets (in shadows) whose total number remains invariant.

2. An ordered sequence of component and FOV events for the time interval $[t_0, t_f]$.

3. Any target motion dynamics (for example, nondeterministic) that may provide additional information about critical events.

From these inputs, the task is to track the evolution of the target distribution and in particular, to estimate at $t = t_f$ the possible number of targets in a given set of shadows. For the rest of this chapter, we focus on two flavors of this problem:

1. A nondeterministic setting in which targets move nondeterministically but critical events are observed without error.

2. A probabilistic setting in which the targets' movement has a probabilistic model and there are imperfect sensors.

2.4 Complexity of Tracking Hidden, Nondeterministically Moving Targets

2.4.1 The nondeterministic problem

In the *nondeterministic* setting, we assume that the targets move nondeterministically. In particular, when a shadow s_i splits into shadows s_j, s_k, the targets inside s_i can split in any possible way as long as the numbers of targets in s_j, s_k are both non-negative. The component events and FOV events are assumed to be observed without error. Given such assumptions, the observation history can be partitioned into two inputs to our filter algorithm:

1. A sequence of shadow and FOV events, and

2. The initial conditions of targets in the shadows at time $t = t_0$.

A typical initial condition for a shadow takes the form

$$\{(a_1, l_1, u_1), (a_2, l_2, u_2), \ldots, (a_k, l_k, u_k)\}, \tag{2.1}$$

in which a_i denotes a subset of target attributes (such as having red color). We assume that elements of the set $\{a_i\}$ for a shadow are pairwise disjoint: If a_i has red color, then no $a_j, j \neq i$ can include targets with the attribute of having red color. The corresponding l_i and u_i denote the lower and upper bounds on the number of targets in the shadow with attribute a_i. For example, we may know that at the beginning, a shadow has six to nine green targets and five targets that may be blue or red. In this case, the initial condition can be written as

$$\{(c = green, 6, 9), (c = blue \text{ or } red, 5, 5)\}.$$

With these inputs, the main task is to determine the lower and upper bounds on the number of targets in any given set of shadows at $t = t_f$ for any combinations of attributes. These obtained bounds are always tight in the sense that any target distribution falling in these bounds is a possible outcome given the initial condition and the observation history.

To make the explanation of the algorithm clear, we first work with a single attribute and ignore FOV events. We also assume for the moment that the initial conditions are tight in the sense that all possible choices of values must be consistent with the later observations (for example, we cannot have an initial condition of four to six targets in a shadow and later find that it is only possible to have two targets in it). We will then show how FOV events, multiple attributes, and other extensions can be handled incrementally.

2.4.2 An integer linear programming (ILP) perspective

For the simplest case, since there is a single attribute and the FOV events are ignored, we can represent the number of targets in a shadow with a single unknown quantity. Let the set of shadows be $\{s_i\}$; we denote the set of corresponding unknowns as $\{x_i\}$. We can write the initial condition for each shadow at $t = t_0$ as two constraints

$$l_i \leq x_i \leq u_i. \tag{2.2}$$

For each event in the sequence of component events, we then obtain one extra constraint of the following form:

$$\begin{aligned} &\text{Appear or disappear:} && x_i = d_i \\ &\text{Split or merge:} && x_i = x_j + x_k. \end{aligned} \tag{2.3}$$

Here we allow that as an appear event happens, d_i targets may hide in s_i at the same time. This is more general than letting $d_i = 0$. To unify notation, we write these in the same way as the initial conditions by letting $l_i = u_i = d_i$. The same applies to the disappear events. Additionally, we have for each shadow s_i, the constraint

$$x_i \geq 0. \tag{2.4}$$

Finally, the task becomes finding the lower and upper bounds of targets for a set of shadows at time $t = t_f$ indexed by $\mathcal{I}$. For the upper bound, we can write the problem as maximizing the sum of the set of unknowns

$$\text{maximize} \sum_{i \in \mathcal{I}} x_i. \tag{2.5}$$

Finding the lower bound then becomes maximizing the set of unknowns not indexed by $\mathcal{I}$ because the total number of targets are preserved. We have obtained an integer linear programming (ILP) problem: All critical events can be expressed using constraints of forms from (2.2), (2.3) and (2.4), with the objective function having the form from (2.5). For example, if we are to express the ILP problem in the *canonical form*, all we need to do is to split each equality constraint (given by (2.3)) into two inequality constraints (for example, $x_i = d_i$ becomes $x_i \leq d_i$ and $x_i \geq d_i$) and multiply all inequality constraints with -1 where necessary ($x_i \leq d_i \Rightarrow -x_i \geq -d_i$). This gives us the ILP problem in canonical form,

$$\text{minimize} \sum_{i \in \mathcal{I}} -x_i, \quad \text{subject to } Ax \geq b, x \geq 0, \tag{2.6}$$

in which A is the constraint coefficient matrix accumulated from initial condition and the critical events; x is the vector of unknowns (one for each shadow). The size of A is determined by the number of shadows and the number of critical events. For additional discussion on ILP modeling, see [90].

2.4.3 Polynomial time solvability of the ILP problem

It is well known that the class of ILP problems is NP-complete in general. It turns out, however, that our ILP problem is not only feasible, but also efficiently solvable. We point out that an actual target tracking problem may require solving more than a pair (upper and lower bounds) of ILP problems as formulated in (2.6). For example, in a fire rescue scenario, it may be necessary to estimate upper and lower bounds on all current shadows individually. Nevertheless, as long as the number of ILP problems are manageable (say, linear with respect to the size of the inputs), the overall problem can also be efficiently solved.

Proposition 3 *A polynomial time algorithm exists for the system described by (2.6).*

As a first step in proving the proposition, we establish the following lemma:

Lemma 4 *The constraint matrix A in (2.6) is totally unimodular.*[1]

Proof. We use induction over the size of square submatrices of A to prove that all such submatrices must have determinant 0 or ± 1. As the base case, every element of A is 0 or ± 1. Suppose that all square submatrices of order n have determinant $0, \pm 1$. Denote these matrices $\mathcal{M}_n$. Suppose there is a square submatrix M of A of order $(n+1)$ with determinant not in $\{0, \pm 1\}$. Every constraint arising from (2.2), (2.3), and (2.4) except $x_i = x_j + x_k$ introduces rows in A with a single ± 1 in them; the rest of the row contains only 0's. If M contains a row arising from these types of constraint then M must have determinant

[1] An integer square matrix A is unimodular if $\det A = \pm 1$. A matrix B is totally unimodular if every non-singular square submatrix of B is unimodular.

$0, \pm 1$ by induction. Suppose not. In this case, all rows of M are introduced by constraint of type $x_i = x_j + x_k$. Each such constraint brings in two rows of A with opposite signs and therefore cannot both appear in M. We can assume that M's first row has coefficients coming from one of the rows introduced by a split event, $x_i = x_j + x_k$. As a first case, let the i, j, k-th columns of A correspond to i', j', k'th columns of M, respectively. To make M's determinant not in $\{0, \pm 1\}$, there needs to be another row in M that contains exactly two nonzero elements among i', j', k'th columns. This is only possible if s_j and s_k merge again, giving a constraint of the form $x_j + x_k = x_l$. We may let this row be the second row in M. This suggests that j', k'-th columns of M are all zeros after the second row; but this gives us that M has determinant 0. The second case is that M includes only two columns of A's i, j, kth columns. It can be checked similarly that M must have determinant 0. ■

Proof of Propsition 3. When the constraint matrix A is totally unimodular and b is a vector of integers, then the minimal faces of the constraint polytope must assume integer coordinates, making the solution of the relaxed linear programming (LP) problem also the solution to the original ILP problem [91]. It is clear that b in (2.6) is integer. Lemma 4 gives us that A is totally unimodular. Therefore, a polynomial time algorithm such as interior point method can be applied to solve (2.6). ■

2.5 The Bipartite Information Space

Although Proposition 3 tells us that the nondeterministic formulation can be solved in polynomial time using generic LP algorithms, it is not clear that these algorithms fully explore the intrinsic structure of the problem at hand. In this section, we briefly review the *information space* (I-space for short, see Chapter 11 of [3] for an introduction) and show how the I-space framework can help with the systematic exploration of the structure of filtering

problems that are combinatorial in nature. For our particular problem, we show that additional information can be discarded from the shadow sequence to yield a further condensed *information state* (I-state). Algorithmic solutions based on max-flow are then introduced, followed by various extensions.

2.5.1 Information space as a guiding principle for task based filtering

As a shadow sequence is extracted from an observation history, a much condensed combinatorial structure is left. This choice is not arbitrary: The general task of tracking unpredictable targets outside the sensor range induces an equivalence relation over the workspace-time space that yields the space of shadows; the evolution of these shadows then gives rise to a space of shadow sequences. In this subsection, we review I-space/I-state concepts and explain how shadow sequences can be viewed as *derived* I-states and the space formed by them a *derived* I-space. We also characterize how I-spaces/I-states, tasks, and filters are closely related.

For any problem, I-space analysis begins with the *history I-space*, $\mathcal{I}_{hist}$, which is essentially the set of all data that robots may ever obtain. Formally, for a time period $[t_0, t_f] \subset T$, a perfect description of everything that occurred would be a *state trajectory* $\tilde{x}_t : [t_0, t_f] \rightarrow X$, in which X is the combined state space of robots and targets. It is impossible to obtain this because not all target positions are known. What is available is the robot's trajectory $\tilde{q}_t = \tau$ and the sensor *observation history* $\tilde{y}_t : [t_0, t_f] \rightarrow Y$, produced by a sensor mapping $h : X \rightarrow Y$, in which Y is the observation space of the sensors. Let the robots also have access to some initial information η_0 at $t = t_0$. The *history I-state* at time t, $\eta_t = (\eta_0, \tilde{q}_t, \tilde{y}_t)$, represents all information available to the robots. The *history I-space* $\mathcal{I}_{hist}$ is the set of all possible history I-states. $\mathcal{I}_{hist}$ is an unwieldy space; it must be greatly reduced if we expect to solve interesting problems. Imagine a robot equipped with a GPS and a video camera moves along some path τ. Without a specific task, the robot will not be able to decide what

information it gathers is useful; therefore, it has to store all of $\tilde{q}_t, \tilde{y}_t$. Even at a relatively low spatial resolution and a frequency of 30 Hz, just keeping the robot's location history and its camera's images in compressed form requires a large amount of storage space, which presently is not generally possible over a long time period.

Once a task is fixed, however, it may become possible to reduce $\mathcal{I}_{hist}$ dramatically. For our specific task of tracking hidden targets in shadows, as we have established in Observation 2, all we need to know is the initial distribution of targets, the component events, and the FOV events. Since targets move unpredictably, other information contained in η_t does not help: The robots' exact location, the shape of the workspace shadows, and what the targets in the FOV are doing are not relevant. Thus, Observation 2 allows us to construct a *derived I-space* $\mathcal{I}_{ss}$, called the *shadow sequence I-space* that discards the irrelevant information. Consider the information contained in $\eta_t = (\eta_0, \tilde{q}_t, \tilde{y}_t)$. To derive $\mathcal{I}_{ss}$, the following reductions are made over $\eta_0, \tilde{q}_t, \tilde{y}_t$:

1. The initial distribution of targets is extracted from η_0.

2. The shadow sequence is extracted via processing $\tilde{q}_t$ and $\tilde{y}_t$.

3. The observation history $\tilde{y}_t$ is compressed so that only critical events and temporal order between these events need to be recorded.

The result from this reduction is the *shadow sequence I-state* η'_t (Fig. 2.7 gives an example) that *lives* in $\mathcal{I}_{ss}$. $\mathcal{I}_{ss}$, as a complete yet more compact representation, immediately reveals much more structure that is intrinsic to our task than $\mathcal{I}_{hist}$ does. From this we observe a general pattern that we exploit: Given $\mathcal{I}_{hist}$ and a task, we try to find one or more sufficient derived I-spaces, and work exclusively in these derived I-spaces. In signal processing, a filter is defined as a device or process that removes from a signal unwanted features [92]. In this sense, the process of extracting shadows from $\tilde{q}_t$ and $\tilde{y}_t$ is exactly a filter. Moreover, $\mathcal{I}_{hist}$ and $\mathcal{I}_{ss}$ are connected through this filter. From this perspective, solving a task becomes finding

the correct I-space, applying the associated filter, and performing additional computation as events happen. For the nondeterministic formulation, we call such filters *combinatorial filters*.

2.5.2 The bipartite information space and the shadow information space

As mentioned at the beginning of this section, generic LP algorithms may not explore the full structure of our problem. One interesting property of our problem is that the distribution of targets in the shadows mimics network commodity flow. Another intrinsic and key property of our problem is that, in many cases, the relative order of component events does not affect the possible target distribution in the shadows. For example, the two shadow sequences in Fig. 2.8 are equivalent: The set of shadows at $t = t_f$ are basically the same.

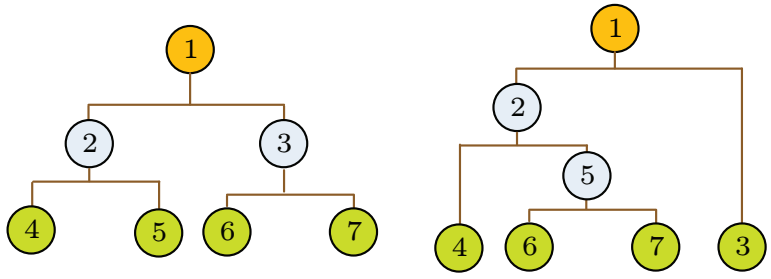


Figure 2.8: Two shadow sequences that are equivalent for task of estimating lower and upper bounds on the number of targets in the shadows at time $t = t_f$.

This allows us to safely discard the intermediate shadows to obtain a more compact I-space $\mathcal{I}_{bip}$, the *bipartite I-space*. The basic idea behind compressing $\mathcal{I}_{ss}$ into $\mathcal{I}_{bip}$ is that, since the robots' sensors cannot obtain information from the shadows as the robots move around, the information that really matters is how shadows from the beginning and the current time are related, while discarding the shadows from intermediate times. By conservation of targets in the environment, the number of targets in the shadows at $t = t_0$ and appeared shadows must be equal that in the shadows at $t = t_f$ and disappeared shadows. This hints toward a bipartite graph structure, which is why we denote the space of such I-states the bipartite

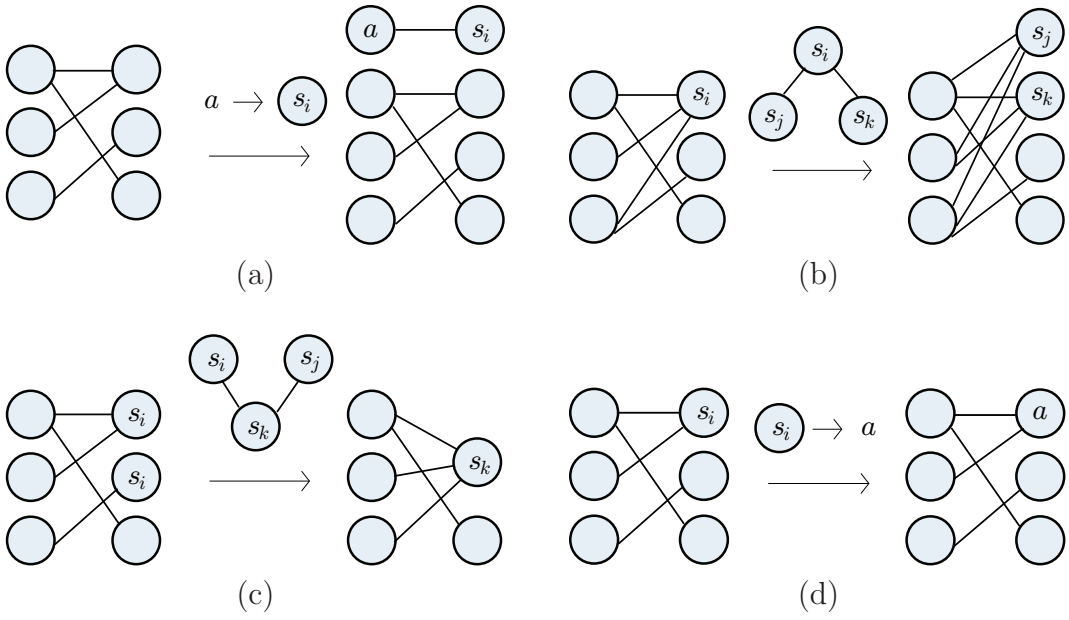


Figure 2.9: Incrementally computing I-states in $\mathcal{I}_{bip}$. (a) An appear component event in which a targets goes into shadow s_i adds two vertices and an edge, with a associated with the left vertex. (b) A split event splits a vertex and all edges pointing to that vertex. (c) A merge event collapses two vertices into one and collapses their ingoing edges. (d) A disappear event in which s_i is revealed to have a targets in it only associates a with the vertex on the right side.

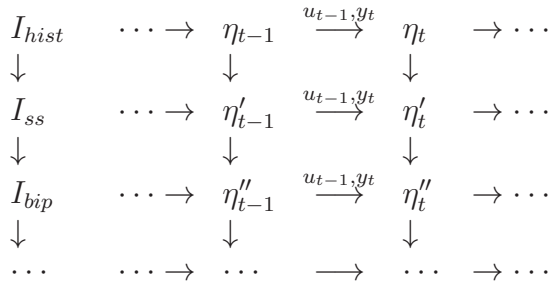


Figure 2.10: The shadow information space structure.

I-space. To do the filtering, the component events are processed individually according to the procedure shown in Fig. 2.9. By the construction of $\mathcal{I}_{ss}$ and $\mathcal{I}_{bip}$, we have shown that $\mathcal{I}_{hist}$, $\mathcal{I}_{ss}$, and $\mathcal{I}_{bip}$ describe the same ILP problem.

Proposition 5 *Given that targets move nondeterministically, information from $\mathcal{I}_{hist}$ and the corresponding $\mathcal{I}_{ss}$, $\mathcal{I}_{bip}$ describe the same ILP problem of the form (2.6).*

PROOF. The invariance from Observation 2 gives us that $\mathcal{I}_{hist}$ and $\mathcal{I}_{ss}$ are equivalent in capturing the distribution of hidden targets. To see that $\mathcal{I}_{ss}$ and $\mathcal{I}_{bip}$ are equivalent, we may consider each hidden target individually: Any flow of a target along a shadow sequence is possible in the corresponding bipartite structure, by construction. ■

A graphical illustration of relationship between I-spaces and I-states, which summarizes the I-space discussion, is given in Fig. 2.10. We point out that such hierarchical structures exist regardless of whether the formulation is nondeterministic or probabilistic; it so happens that for our filtering problem, the nondeterministic formulation leads to one more level of natural structure than the probabilistic formulation (see Section 3.2).

Another practical implication of the I-space structure is that, if the task is known before the observation history is obtained, the derived I-state can be obtained "online." Most of η_t can be discarded once we have η'_t, and η'_{t+1} can then be obtained from η'_t and $q_{t,t+1}, y_{t,t+1}$, which are the information accumulated during the time interval $(t, t+1]$. This is illustrated

for $\mathcal{I}_{hist}$, $\mathcal{I}_{ss}$, and $\mathcal{I}_{bip}$ in Fig. 2.10. Computationally, the shadow sequence can be obtained by storing and processing only immediate history. At a given time t, since our analysis establishes a 1-1 correspondence between workspace-time shadows and workspace shadows at t, we only need to look at observation history between $t-t_\epsilon$ ($t_\epsilon > 0$ is a small real number) and t to detect component events. That is, workspace-time shadows, which determines the shadow sequence, can be recovered from workspace shadows. Similar techniques apply to the detection of FOV events.

CHAPTER 3

COMBINATORIAL FILTERS FOR SHADOW INFORMATION SPACES

With the three level shadow information space structure obtained in Chapter 2, we have essentially built a filtering process for maintaining sufficient information for estimating the number of targets hidden in a shadow (or a set of shadows). In particular, the $\mathcal{I}_{bip}$ states, combined with an initial condition, allow us to compute these numbers at any given time. In this chapter, we first show how this computation can be efficiently carried out for nondeterministically moving targets. Then, we show that the shadow information space is equally applicable to probabilistically moving targets.

3.1 Flow-Based Combinatorial Filters for Tracking Nondeterministically Moving Targets

3.1.1 Tracking targets as a max-flow problem

With the bipartite I-state structure, we are ready to illustrate the complete combinatorial filtering process with a concrete example (the procedure was first introduced in [71]). After obtaining the bipartite structure, the rest of the algorithm is nothing more than applying a maximum flow subroutine (such as Edmonds-Karp) [93]. For the environment given in Fig. 3.1(a), a visibility cell decomposition procedure [94] will give us the shadow sequence I-state in Fig. 3.1(b). Applying the $\mathcal{I}_{bip}$ filter then gives us the bipartite graph in Fig. 3.1(c). Note that each shadow becomes a vertex (sometimes two vertices) of the bipartite graph. Once the bipartite graph is constructed, the task of determining lower and upper bounds on shadows

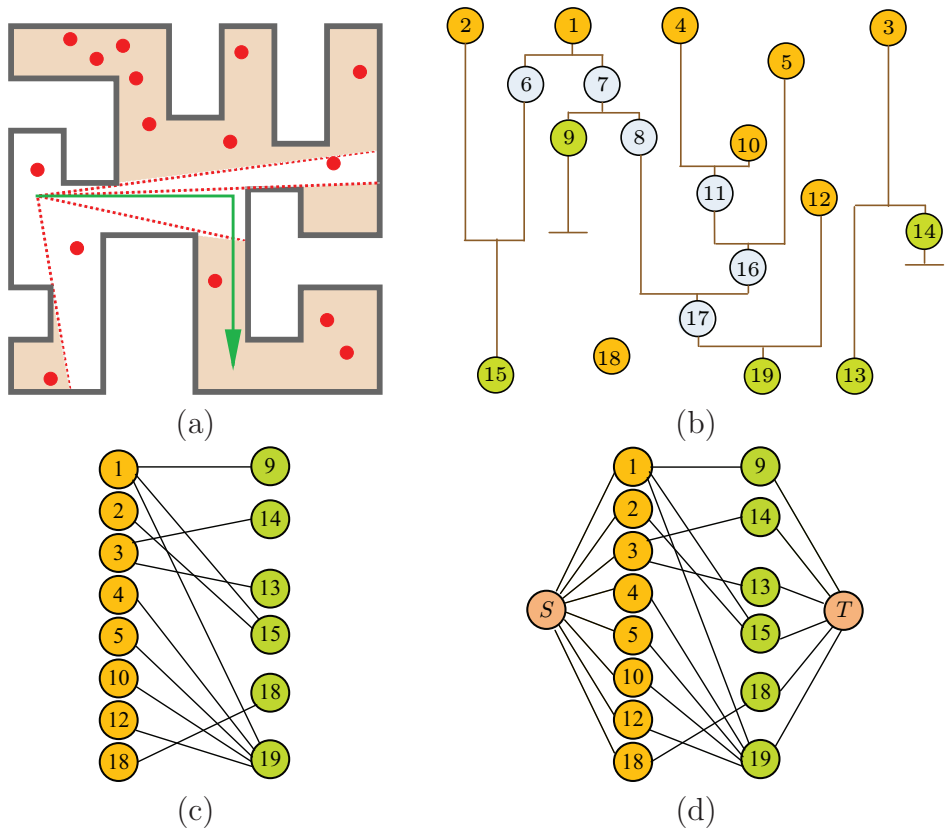


Figure 3.1: Intermediate steps in the computations of target bounds. (a) A 2D office-like environment. A single robot follows the green path. Red dots are illustrations of possible targets in the environment. (b) The shadow sequence I-state for the environment and path. The orange indexed shadows are these at $t = t_0$ or appearing; the green ones are these at $t = t_f$ or disappearing. Shadow s_{18} are both appearing and existing at $t = t_f$. (c) The bipartite I-state. (d) Augmented graph for running max-flow algorithm.

at $t = t_f$ can be transformed into a max-flow problem. To achieve this, we first augment the graph by adding a source vertex S and sink vertex T. An edge is added between S and each shadow at $t = t_0$ as well as each appeared shadow, and an edge is added between T and each shadow at $t = t_f$ as well as each disappeared shadow. The end result of doing this to the graph in Fig. 3.1(c) is Fig. 3.1(d).

After obtaining the extended graph, capacities need to be assigned to edges of the graph before running max-flow. Let $e(v_1, v_2)$ be an edge in the graph from vertex v_1 to vertex v_2, and denote the capacity and flow on the edge as $c(v_1, v_2), f(v_1, v_2)$, respectively. Suppose

that we want to obtain the upper bound on the number of targets in shadow s_{19}. The edges of the original bipartite graph will always have infinite capacities, which we do not mention again. For each edge between S and a shadow indexed i, let $c(S, i) = u_i$. In our example these indices are 1-5, 10, 12, and 18. For each edge between a disappearing shadow indexed i, and T, let $c(i, T) = l_i$. These are 9 and 14 in our example. Since we want as many targets to go to s_{19} as possible, we let $c(i, T) = 0$ for $i = 13, 15, 18$ and $c(19, T) = +\infty$. After running the max-flow algorithm, the maximum possible number of targets that can end up in s_{19} is given by

$$f(19, T) + \sum_i f(i, T) - \sum_i c(i, T), \tag{3.1}$$

in which the summations are over indices of disappearing shadows. We need to consider disappearing shadows since these shadows should have flow equal to their capacity, which is not guaranteed by a max-flow algorithm. Filling in numbers into this example, assuming that the input lower/upper bound pairs for shadows 1-5, 9, 10, 12, 14, 18 are $(2, 4)$, $(0, 3)$, $(5, 5)$, $(2, 6)$, $(4, 5)$, $(2, 3)$, $(1, 3)$, $(3, 8)$, $(2, 4)$, $(5, 7)$, respectively, then (3.1) gives us that shadow 19 can have at most 24 targets. If we instead want the lower bound on the number of targets in s_{19}, we should let $c(S, i) = l_i$, $c(i, T) = l_i$ for $i = 9, 14$, $c(i, T) = +\infty$ for $i = 13, 15, 18$ and $c(19, T) = 0$. After running the max-flow algorithm, s_{19}'s lower bound is given by

$$\sum_i c(S, i) - \sum_j f(j, T), \tag{3.2}$$

in which the first summation is over all shadows connected to S and the second summation is over all shadows connected to T. Using the earlier numbers, this minimum is 10 for shadow 19. The same procedure applies to an arbitrary set of shadows.

3.1.2 Incorporating FOV events

In the nondeterministic setting, there is no null FOV event. As mentioned in Observation 2, exit and enter FOV events can be handled by converting them into component events. To convert an enter FOV event of shadow s_i into component events, we simply create an appear component event of a single target and then merge the newly created shadow into s_i. Similarly, an exit FOV event can be converted into a split component event followed by a disappear component event. The rest of the algorithm stays the same. The problem is, however, if there is a large number of FOV events compared to the number of component events, this approach will slow down later steps of the algorithm since it will create two component events per FOV event. Fortunately, there is no reason to handle each FOV event individually; since each FOV event is associated with some shadow, we can group them based on this association. The only caveat is that we cannot just group all FOV events for one shadow into a single *batch* FOV event as this can introduce information loss. For example, if e_x, e_x, e_e, e_e happens to shadow s_i, this is not equivalent to nothing has happened. We know that s_i must have at least two targets in it originally (a "surplus"). On the other hand, the just mentioned surplus and net target flow are the only two pieces of information that FOV events of a shadow give us; hence up to two batch FOV events can summarize all information contained in all FOV events for a given shadow. Let $\langle e_j \rangle$ be the sequence of FOV events for a shadow s_i in which e_j is either e_e or e_x, we build a counter to track the surplus of s_i as $d_{\min} = \min\{d_j\}$, with d_j defined as

$$d_j = \begin{cases} d_{j-1} + 1 & \text{if } e_j = e_e \\ d_{j-1} - 1 & \text{if } e_j = e_x \\ 0 & \text{if } j = 0. \end{cases}$$

Let d_{tot} be d_j for the last j, the net target flow from FOV events. We have four cases. If $d_{\min} = d_{\text{tot}} = 0$, we do nothing. If $d_{\min} \geq 0$ and $d_{\text{tot}} > 0$, we only need to create one batch

enter FOV event for s_i with d_{tot} number of targets. If $d_{\min} < 0$ and $d_{\text{tot}} = d_{\min}$, we only need to create one batch exit FOV event with $|d_{\min}|$ number of targets. In the last case, we need to create one batch exit FOV event with $|d_{\min}|$ number of targets and then an enter FOV event with $d_{\text{tot}} - d_{\min}$ number of targets. We can then apply the naive approach from the beginning of this subsection to convert these batch FOV events into component events. With this construction, we never need to handle more than $5n$ events in which n is the maximum number of shadows.

3.1.3 Solving a variety of other tasks

The ability to obtain lower and upper bounds of the number of targets hiding inside a shadow easily extends to other useful tasks. We briefly cover a few of these variations.

Refining initial bounds

Max-flow computations can also be used to refine the lower and upper bounds from initial conditions if they are not tight. To get a refined lower bound for a shadow at $t = t_0$, say s_1 from Fig. 3.1(b), let $c(S, 1) = l_1$, $c(S, i) = u_i$ for $i \neq 1$, $c(i, T) = u_i$ for disappearing shadows, and $c(i, T) = 0$ for the rest. After running max-flow on this network, a tighter lower bound, if there is one, is given by

$$l_1' = l_1 + \sum_i c(i, T) - \sum_j f(j, T). \tag{3.3}$$

The summations are done similar to that of (3.2). To refine u_1, let $c(S, 1) = u_1$, $c(S, i) = l_i$ for $i \neq 1$, $c(i, T) = u_i$ for disappearing shadows and $c(i, T) = +\infty$. After running max-flow,

$$u_1' = f(S, 1). \tag{3.4}$$

This procedure also applies to a set of shadows.

Counting

In this case, the total number of targets, n, is unknown. For determining n, the lower and upper bounds on each shadow at $t = t_0$ are set as $l_i = 0, u_i = +\infty$. As new component or FOV events are observed by the robots moving in the environment, the previous procedure is run to keep refining the initial bounds. Once we have $l_i = u_i$ for each initial condition, n has been determined. Note that if the free space is not completely explored, then the upper bound remains at infinity. Another instance of counting is knowing n. For example, in a wild animal preserve, it may be required that the total number of a species is verified periodically. This reduces to the problem of being given n and wanting to account for all of them. To verify the count, we can keep track of the lower bounds on the total number of targets, and if the number agrees with n, then the task has been accomplished.

Pursuit-evasion

Suppose there is a single evader and the task is to determine where it might be. In this case, $l_i = 0, u_i = 1$ for each shadow at $t = t_0$. There are three possibilities for each shadow at $t = t_f$: (1) $l_i = u_i = 0$ (the evader is not in s_i), (2) $l_i = u_i = 1$ (the evader is definitely in s_i), and (3) $l_i = 0, u_i = 1$ (the evader may or may not be in s_i). Note that this is a passive version of the pursuit-evasion problem. We do not determine a trajectory that is guaranteed to detect the evader. In general, this problem is NP-hard [23]. Nevertheless, the calculation method proposed in this chapter can be used with heuristic search techniques (or even human operators) to correctly maintain the status of the pursuit.

3.1.4 Incorporating distinguishability

So far we only considered the case of a single attribute, which is the fully indistinguishable case. What about multiple attributes? We consider two important cases of distinguishability based on whether attributes get mixed up or not. If attributes are not intertwined, i.e., each

a_i in (2.1) is a single attribute, it is straightforward to see that for m attributes, all we need to do is to run the algorithm for a single attribute m times, once for each attribute. Additional computation can then be performed to calculate more complicated combinations. For example, if we want the lower and upper bounds on the number of all targets for a shadow, then we can simply add up individual lower and upper bounds.

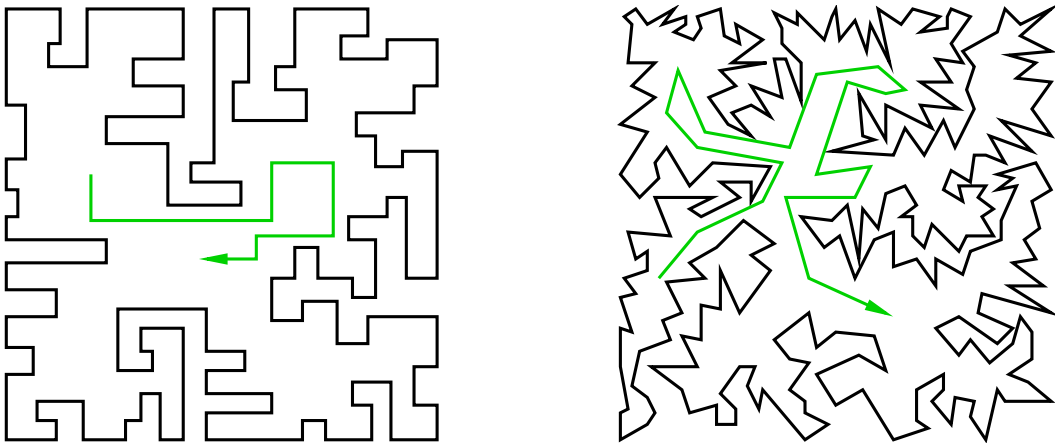
For the second case in which we may have multiple attributes for some a_i, the basic flow-based approach does not work. Using the example from Fig. 3.1, suppose that there are two teams, red and blue, and the initial conditions of shadows at $t = t_0$ are of the form (red or $blue, l_i, u_i$). Suppose that we want to get the lower and upper bound of the number of targets in s_{19} again. For lower bounds, four computations are needed. First, we set red capacities to 0 and blue capacities to l_i for all edges starting from S. The capacities for each color for edges ending in T are set as before. Running two max-flow computations, one for red and one for blue, gives us one possible lower bound l_{r1}, l_{b1}. Switching red and blue and repeating the basic flow-based procedure gives us another lower bound l_{r2}, l_{b2}. We should have $l_{r1} + l_{b1} = l_{r2} + l_{b2}$. The lower bound on s_{19} is then $l_{r1} + l_{b1}$ red or blue targets with between l_{r1} and l_{r_2} red targets. The upper bound can be obtained similarly.

3.1.5 Simulation results and complexity analysis

For the nondeterministic case, we implemented and tested the algorithms[1] for a single robot that moves in a simply connected polygonal region in $\mathbb{R}^2$ using an omnidirectional visibility sensor. For such environment, the shadows are completely characterized by bitangents and inflections. When the environment is known and the robot can localize itself, efficient 2D cell decomposition algorithms can be readily applied to obtain the sequence of shadows for each location of the robot, allowing the shadows to be continuously tracked. This setup also

[1]The simulation programs were developed adhering to the Java 1.6 language standard under the Eclipse environment. The computations were performed on a workstation with an Intel Core 2 Quad processor running at 3.0 GHz. The JavaVM has a maximum memory of 1.5GB. Although many parts of our algorithm can be easily parallelized, no multi-threading was used in this implementation.

enables us to construct an oracle (not available to the algorithm) for distributing targets inside the free space to simulate their nondeterministic behavior. For max-flow, we implemented the $O(VE^2)$-time Edmonds-Karp max-flow algorithm [93], in which V and E are the numbers of vertices and edges in the flow graph, respectively.



(a) (b)

Figure 3.2: Complicated examples that were used to test our approach. The given robot trajectories are shown as green lines in the direction of the arrow.

For the environment in Figure 3.2(a), the trajectory generates 85 component events. Our oracle randomly distributed 100 targets in the free space as the component events occur. This setting yields a bipartite graph that has 41 vertices and 60 edges. Calculating the lower and upper bounds for the 18 final shadows for a single team took 0.1 seconds. The second free space, shown in Figure 3.2(b), has 385 component events, 491 total shadows, 124 vertices in the bipartite graph with 339 edges. The example involves a million targets with five teams that intersperse. The bounds on the 12 final shadow components for all five teams were computed under one second.

The inputs to the base algorithm (single attribute, no FOV events) are: (1) a sequence of n shadows, and (2) the initial condition which takes the form of a pair of lower and upper bounds for each shadow at $t = t_0$. In the worst case, there are $O(n)$ vertices and $O(n^2)$ edges in the bipartite graph. Edmonds-Karp max-flow then gives us $O(n^5)$ running time

in the worst case. Applying a push-relabel algorithm with FIFO vertex selection rule will cut the running time to $O(n^3)$ [95]. Adding FOV events does not increase time complexity asymptotically, as discussed in Subsection 3.1.2. Adding partial distinguishability, on the other hand, will introduce another input parameter m, the number of teams, that contributes linearly to time complexity. The typical worst case running time for the nondeterministic case is then $O(n^3m)$. The number of targets in the system does not directly affect the performance.

3.2 Imperfect Sensors, Probabilistic Events, and Bayesian Filters

Now consider the case of *probabilistic* uncertainty. So far we have assumed that shadows and events are always reported without any error, which is unrealistic in practice. For detecting shadows, we already mentioned that true sensing range may be unavailable for some sensors and sometimes it is simply computationally impractical to obtain the exact visible/shadow region. However, if we settle for partial correctness, then probabilistic models can be applied. For example, when we deal with sensor networks, conservative, probabilistic estimates of sensing range may suffice.

The same principle applies to FOV events. For each of the three FOV events, we assume that the sensors on the robots may correctly observe it or mistake it for the other two events. An enter event for a component may be reported by the sensor as an enter, exit, or null event; the same applies to exit and null events. That is, the sensor mapping is given by $h : E_{FOV} \to Y_{FOV}$, with Y_{FOV} being the set of *FOV observation*s

$$Y_{FOV} = \{y_e, y_x, y_n\},$$

in which y_e, y_x, and y_n are enter, exit, and null observations. The map h can be deterministic, nondeterministic, or probabilistic. In this section, the case of a probabilistic FOV event-

sensor mapping is investigated, together with the assumption that the dynamics of a split event is provided.

Before moving on, we introduce some notations to facilitate the discussion of the probabilistic formulation. We use s_i to denote the shadow with label i, as well as the random variable for that shadow in the joint/multivariate distribution. For shadows $s_1, \ldots, s_n$, the joint distribution is then $P(s_1, \ldots, s_n)$, in which a specific entry is $P(s_1 = x_1, \ldots, s_n = x_n) \in [0, 1]$. In writing formulas and outlining algorithms, we shorten the repeated variables to "..." on both the left-hand side (LHS) and the right-hand side (RHS) of an expression. In such cases, the combined "..." on the LHS and RHS denote the same set of random variables. For example, $P(s_1, s_2, s_3, s_4) = P(s_1, s_2, s_k, s_3, s_4)$ is shortened to $P(\ldots) = P(\ldots, s_k, \ldots)$.

3.2.1 A probabilistic formulation

In the basic setup, besides the availability of a sequence of component and FOV event observations (e.g. Fig. 2.7), the following assumptions are made:

1. Component events are observed without error.

2. Targets are indistinguishable. The initial condition is given as a joint probability distribution $P(s_1, \ldots, s_n)$ of targets in the n shadows at $t = t_0$.

3. When a split component event happens, a probabilistic *split rule* decides how the targets should redistribute.

4. Observations of FOV events follows distribution given by $P(\mathbf{e} = e|\mathbf{y} = y), e \in E_{FOV}, y \in Y_{FOV}$.

After general algorithms are presented, we discuss extensions relaxing the first two assumptions. The last two assumptions can be satisfied by collecting and analyzing sensor data from the same environment; the necessity of these two assumptions will become self-evident

shortly. Given these assumptions, we want to obtain the target distribution in the m shadows, $P(s'_1, \ldots, s'_m)$, at time $t = t_f$.

The resulting joint probability distribution is useful in solving many decision making problems; for example, in a fire evacuation scenario, knowing the the expected number of people trapped in various parts (shadows) of a building (possibly estimated through observations from infrared beam sensors or security cameras), firefighters can better decide which region of the building should be given priority when they look around. The expected number of people in each shadow is readily available from the joint probability distribution.

3.2.2 Processing component events

To understand how observations affect target distributions in a probabilistic setting, let us first look at the component events (we do not distinguish between events and observations for these since they are the same by assumption). Among the four types of component events, split and disappear events are more important than appear and merge events.

1. **Split.** A split event introduces more uncertainty. As a shadow splits into two disjoint shadows, the probability masses in the newly spawned shadows cannot be predicted without additional information because the sensors can not see what happens within the shadow region during a split event. The issue is resolved by the introduction of a *split rule*, obtained from supporting data or an oracle, which dictates how the originating shadow's probability mass should be redistributed. For example, statistical data may support that the number of targets in the child shadows are proportional to their respective areas.

2. **Disappear.** When a shadow disappears, the targets hiding behind it are revealed. This information can be used to update our belief about the target distribution by eliminating some improbable distributions of targets. In particular, it can reduce the

uncertainty created by split events. For example, suppose that a shadow s_i, having d_i targets in it (with 100% probability), splits into shadows s_j and s_k. It is possible that s_j has 0 to d_i targets in it, as does s_k. However, if s_k later disappears to reveal d_k targets in it and no other events happen to s_j and s_k, then s_j must have exactly $d_i - d_k$ targets in it. In general, assuming that shadow s_k disappears with a target distriubtion $P(s_k)$, the update rule is given by

$$P'(s_1 = x_1, \ldots, s_n = x_n) \propto \sum P(s_1 = x_1, \ldots, s_k = x_k, \ldots, s_n = x_n) P(s_k = x_k),$$

in which the summation is over all joint probability entries of $P(s_1, \ldots, s_n)$ such that $s_k = x_k$. Normalization is required.

3. **Appear.** An appearing shadow s_k, with distribution $P(s_k)$, can be joined with the rest via combining the independent distributions $P(s_k)$ with $P(s_1, \ldots, s_n)$:

$$P'(s_1 = x_1, \ldots, s_n = x_n, s_k = x_k) = P(s_1 = x_1, \ldots, s_n = x_n) P(s_k = x_k).$$

4. **Merge.** In this case, two probability masses are collapsed. We simply collect the joint distribution to form a single one,

$$P'(\ldots, s_k = x_k) = \sum_{x_i + x_j = x_k} P(\ldots, s_i = x_i, \ldots, s_j = x_j, \ldots),$$

in which s_k is the merged shadow of shadows s_i and s_j. The operations involved in a merge is essentially that of a marginalization. A detailed example is given in Table 3.1 in which the original shadows are s_1, s_2, s_3 and s_2, s_3 merge to form shadow s_4.

Table 3.1: Propagating probability masses: An example.

before merge	$P(s_1 = 1, s_2 = 1, s_3 = 4) = 0.2$ $P(s_1 = 1, s_2 = 2, s_3 = 3) = 0.2$ $P(s_1 = 1, s_2 = 3, s_3 = 2) = 0.2$ $P(s_1 = 2, s_2 = 1, s_3 = 3) = 0.2$ $P(s_1 = 2, s_2 = 2, s_3 = 2) = 0.2$
after merge	$P(s_1 = 1, s_4 = 5) = 0.2 + 0.2 + 0.2 = 0.6$ $P(s_1 = 2, s_4 = 4) = 0.2 + 0.2 = 0.4$

3.2.3 Processing FOV events and observations

Shifting to FOV events, we observe that an enter event only affects the shadow being entered by increasing the expected number of targets in the shadow. If there is a single shadow s and an enter event happens, we merely update $P(s = d_i) = p_i$ to $P(s = d_i+1) = p_i$. On the other hand, an exit event does the opposite and we change $P(s = d_i) = p_i$ to $P(s = d_i - 1) = p_i$. A complication arises here: If shadow s_i splits into shadows s_j, s_k and an e_x event happens to shadow s_j, it suggests that it is impossible for s_j to have 0 target before the e_x event. The affected probability mass needs to be removed and the remaining values renormalized. The null event does not change the target distribution.

Now, to propagate a probability mass through an FOV observation, y, we essentially break the entry into three pieces according to above rules, multiplying each resulting entries with the probability $P(\mathbf{e} = e_e \mid \mathbf{y} = y), P(\mathbf{e} = e_x \mid \mathbf{y} = y)$, and $P(\mathbf{e} = e_n \mid \mathbf{y} = y)$, respectively. If an enter event is not possible for the observation, the two remaining entries are renormalized.

3.2.4 Balancing between estimation accuracy and computation

As we try to make a general method for handling critical events without modeling under any particular distribution or split rule, another issue arises: The number of targets is discrete, but discrete joint probability distributions can require a large amount of space and time to

process when there are a large number of targets, shadows, and events in the system. On the other hand, if we attempt to use nice discrete probability mass functions or continuous probability density functions for approximation, it becomes problematic when there are only a few targets and events. In such cases, the loss of estimation accuracy may become unacceptably large after only a short sequence of events.

For sequential probabilistic estimation tools such as sequential Monte Carlo method, there is an intrinsic trade-off between accuracy and the amount of computation one can afford. In our problem, this balance depends on the number of targets in the system and the number of events that happen, which can be roughly partitioned into three categories:

1. There are only a few targets and events. In this case, approximation errors early in the procedure can grow very fast as computation is carried out. On the other hand, because the combinatorial choices are limited, this case can be treated with high fidelity, with the only assumption being good split rule and sensor statistics.

2. There are a few targets but a large number of events. Propagating the probability mass through many events will likely accumulate significant errors when there are only a few targets, unless an extremely reliable split rule and sensor statistics are available. If this is the case, the method for the first category will perform well. Otherwise, a probabilistic approach may give results that are far off from the true distribution; the nondeterministic approach mentioned earlier would be a better alternative.

3. There are many targets in the system, in which case there is more freedom in making simplifications without dramatically altering the outcome.

To provide an idea of what we mean by "a few" and "many," our non-optimized Java implementation for the first category can handle tens of targets and events, beyond which the JavaVM will run out of memory. In comparison, the heuristics employed for the third category can handle thousands of targets and events. In the next two subsections, we give

detailed analysis of the first and third categories.

3.2.5 Accurately propagating probability masses

The first algorithm we introduce in this section is one that solves the probabilistic formulation from Subsection 3.2.1 exactly. As events happen, the probability mass, $P(s_1, \ldots, s_n)$, is updated according to Algorithms 1 and 2 based on earlier analysis, in which the *observation* data structure is defined in Table 3.2.

Table 3.2: The *observation* data structure used in Algorithm 1.

event	event type, can be one of *appear, disappear, split, merge* component events and *enter, exit, null* FOV events
s_s	the originating shadow in a split event
s_{s1}	the first new shadow after a split event
s_{s2}	the second new shadow in a split event
s_{m1}	the first shadow in a merge event
s_{m2}	the second shadow in a merge event
s_m	the newly merged shadow
s_e	the newly appeared shadow from an appear event
$P(s_e = n_e)$	probability that s_e contains n_e targets
s_v	the disappearing shadow in a disappear event
$P(s_v = n_v)$	probability that s_v contains n_v targets

As a demonstration, we work through the observation sequence given by Fig. 3.3, with the following assumptions:

1. Initially there are 2 targets each in shadow s_1, s_2.

2. The split rule is that each target has 0.5 probability of going into each of the two split shadows.

3. There is no null event or observation, with the true positive rate for any observation being $p = 0.9$.

Table 3.3: Running the exact algorithm on a simple example.

observation	probability masses
initial	$P(s_1 = 2, s_2 = 2) = 1$
y_x, s_1	$P(s_1 = 1, s_2 = 2) = 0.9$ $P(s_1 = 3, s_2 = 2) = 0.1$
split, $s_2 \rightarrow s_3, s_4$	$P(s_1 = 1, s_3 = 0, s_4 = 2) = 0.9 * 0.25 = 0.225$ $P(s_1 = 1, s_3 = 1, s_4 = 1) = 0.9 * 0.5 = 0.45$ $P(s_1 = 1, s_3 = 2, s_4 = 0) = 0.9 * 0.25 = 0.225$ $P(s_1 = 3, s_3 = 0, s_4 = 2) = 0.1 * 0.25 = 0.025$ $P(s_1 = 3, s_3 = 1, s_4 = 1) = 0.1 * 0.5 = 0.05$ $P(s_1 = 3, s_3 = 2, s_4 = 0) = 0.1 * 0.25 = 0.025$
y_e, s_3	$P(s_1 = 1, s_3 = 1, s_4 = 2) = 0.225$ $P(s_1 = 1, s_3 = 0, s_4 = 1) = 0.45 * 0.1 = 0.045$ $P(s_1 = 1, s_3 = 2, s_4 = 1) = 0.45 * 0.9 = 0.405$ $P(s_1 = 1, s_3 = 1, s_4 = 0) = 0.225 * 0.1 = 0.0225$ $P(s_1 = 1, s_3 = 3, s_4 = 0) = 0.225 * 0.9 = 0.2025$ $P(s_1 = 3, s_3 = 1, s_4 = 2) = 0.025$ $P(s_1 = 3, s_3 = 0, s_4 = 1) = 0.05 * 0.1 = 0.005$ $P(s_1 = 3, s_3 = 2, s_4 = 1) = 0.05 * 0.9 = 0.045$ $P(s_1 = 3, s_3 = 1, s_4 = 0) = 0.025 * 0.1 = 0.0025$ $P(s_1 = 3, s_3 = 3, s_4 = 0) = 0.025 * 0.9 = 0.0225$
$merge, s_1, s_3 \rightarrow s_5$	$P(s_4 = 2, s_5 = 2) = 0.225$ $P(s_4 = 1, s_5 = 1) = 0.045$ $P(s_4 = 1, s_5 = 3) = 0.405 + 0.005 = 0.41$ $P(s_4 = 0, s_5 = 2) = 0.0225$ $P(s_4 = 0, s_5 = 4) = 0.2025 + 0.0025 = 0.205$ $P(s_4 = 2, s_5 = 4) = 0.025$ $P(s_4 = 1, s_5 = 5) = 0.045$ $P(s_4 = 0, s_5 = 6) = 0.0225$
disappear, s_5	$P(s_4 = 0) = 0.0769 = 0.0225 * 0.5/((0.0225 + 0.045 + 0.225) * 0.5)$ $P(s_4 = 1) = 0.1538$ $P(s_4 = 2) = 0.7692$

Algorithm 1 ProcessProbabilityMass

Input: $P(s_1, \ldots, s_n)$, the initial target distribution
Q, the queue of observation sequences
a **split rule**
$P(\mathbf{e} \mid \mathbf{y})$, the sensor statistics
Output: the target distribution after all observations

foreach event *observation* o in Q
  switch($o.event$)
  case *appear*:
    update all $P(s_1 = x_1, \ldots, s_n = x_n) = p_j$ entries to
      $P(s_1 = x_1, \ldots, s_n = x_n, o.s_e = n_e) = p_j * P(o.s_e = n_e)$
  case *disappear*:
    set $P(s_1 = x_1, \ldots, s_n = x_n)$ to
      $\sum P(s_1 = x_1, \ldots, o.s_v = n_v, \ldots, s_n = x_n) * P(o.s_v = n_v)$
    remove stale entries and renormalize the probability masses
  case *split*:
    add two new shadows $o.s_{s1}$, $o.s_{s2}$
    split prob. mass in $o.s_s$ into $o.s_{s1}$, $o.s_{s2}$ by **split rule**
  case *merge*:
    add a new shadow $o.s_m$ and set $P(\ldots, o.s_m = n)$ to
      $\sum_{n_1+n_2=n} P(\ldots, o.s_{m1} = n_1, \ldots, o.s_{m2} = n_2, \ldots)$
  case *enter, exit, null*:
    call ProcessFOVEvent
return the updated target distribution

4. $a_5 = 1$ with probability 0.5 and $a_5 = 2$ with probability 0.5.

The extra assumptions are made so that the calculation of the probability mass entries is limited and the entries can be listed in a table. The iterative processing of observations is shown in Table 3.3. The distribution is represented using a table of joint probabilities, which is always practical when there are not too many targets and events. Renormalization is performed in the third step for the first and sixth entries, as well as in the last step. In the merge step, the third and seventh entries from previous step are combined, as are the fifth and ninth entries. A graphical illustration of the probability masses during each step of the run is given in Fig. 3.4. Note that the dimensions change as component events happen.

To verify the correctness of the outcome, Monte Carlo trials are also run, in which individual targets are propagated through the observation one by one. Since it is not an

Algorithm 2 ProcessFOVEvent

Input: $P(s_1, \ldots, s_n)$, the target distribution
$P(\mathbf{e} \mid \mathbf{y})$, the sensor statistics
$y \in \{y_e, y_x, y_n\}$, the FOV observation
s_i, the affected shadow
Output: the target distribution after the observation

1: **foreach** $P(\ldots, s_i = x_i, \ldots) = p_j$ entry in the distribution
2: **let** $P'(\ldots, s_i = x_i + 1, \ldots) = p_j * P(\mathbf{e} = e_e \mid \mathbf{y} = y)$
3: **let** $P''(\ldots, s_i = x_i, \ldots) = p_j * P(\mathbf{e} = e_n \mid \mathbf{y} = y)$
4: **if** $x_i > 0$
5: **let** $P'''(\ldots, s_i = x_i - 1, \ldots) = p_j * P(\mathbf{e} = e_x \mid \mathbf{y} = y)$
6: **else**
7: normalize P', P'' such that $P' + P'' = p_j$
8: remove $P(\ldots, s_i = j, \ldots) = p_j$ entry
9: store entries P', P'' and P''' if applicable
10: **return** the updated target distribution

exact method, we leave the details of it to the next subsection. After 1000 successful random trials (this is the number of trials used for all Monte Carlo simulations in this chapter), we obtained $P(s_4 = 0) = 0.079, P(s_4 = 1) = 0.154, P(s_4 = 2) = 0.767$, which matches closely the results of the exact algorithm.

3.2.6 Efficiently propagating probability masses

Although the algorithm ProcessProbabilityMass is exact, its performance directly depends on the number of probability mass entries of a particular problem. When there are few targets and events, this is not a problem; but what if this is not the case? For a slightly more complicated event observation sequence (Fig. 3.3), with five targets each in shadow s_1 and s_2 to start, 135 joint probability table entries are obtained before the merge step, as shown in Fig. 3.5. The probability mass entries increase rapidly because of the split events and the FOV events. For a split event, if the originating shadow contains up to n targets, then the number of probability mass entries can multiply by up to a factor of $n + 1$. For FOV observations, each has certain probability to be enter, exit, and null events, which may

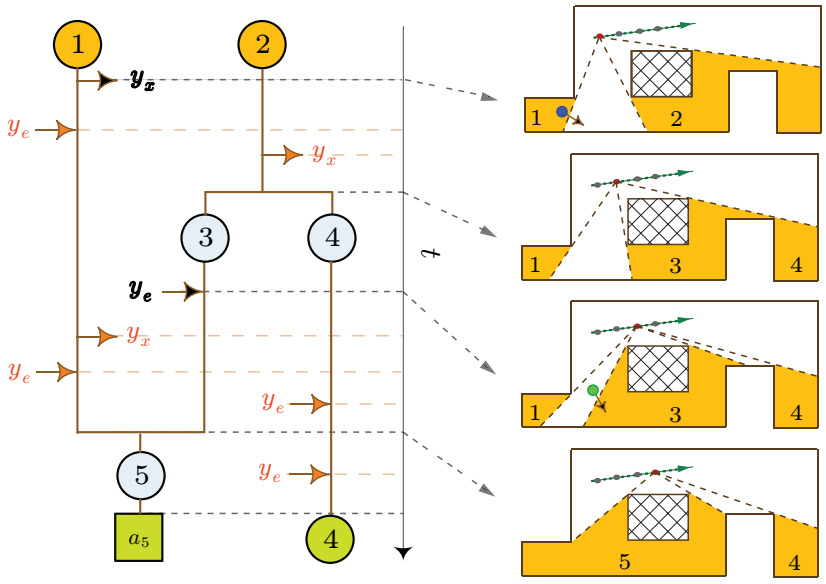


Figure 3.3: A simple event observation sequence is generated (on the left, with only two FOV observations marked with bold faced font) when a robot carrying omni-directional, infinite range sensor follows the dotted path in a polygonal environment with a hole (the four figures on the right). The last event, disappearing of shadow s_5, is not shown on the right; we note that additional resource is needed to make s_5 disappear (say, a sub search team). A slightly more complicated sequence is also possible with six additional FOV observations (on the left, marked with lightened font).

cause the number of probability mass entries to triple in the worst case. Therefore, as the number of targets and events increase, the space required to store the probability masses may grow exponentially. Since processing each observation requires going through all the entries, computation time will also explode, which means that the exact algorithm will not work efficiently. On the bright side, when a large number of probability mass entries are present, some of these entries must have very low weights; making approximations by trimming away these low probability entries is unlikely to greatly affect the final target distribution.

Monte Carlo trials

Since our task is to probabilistically track targets, sequential Monte Carlo methods are a natural choice. As a first heuristic, we perform simple trials such that each trial starts with the initial distribution of targets. These targets are propagated through the event observa-

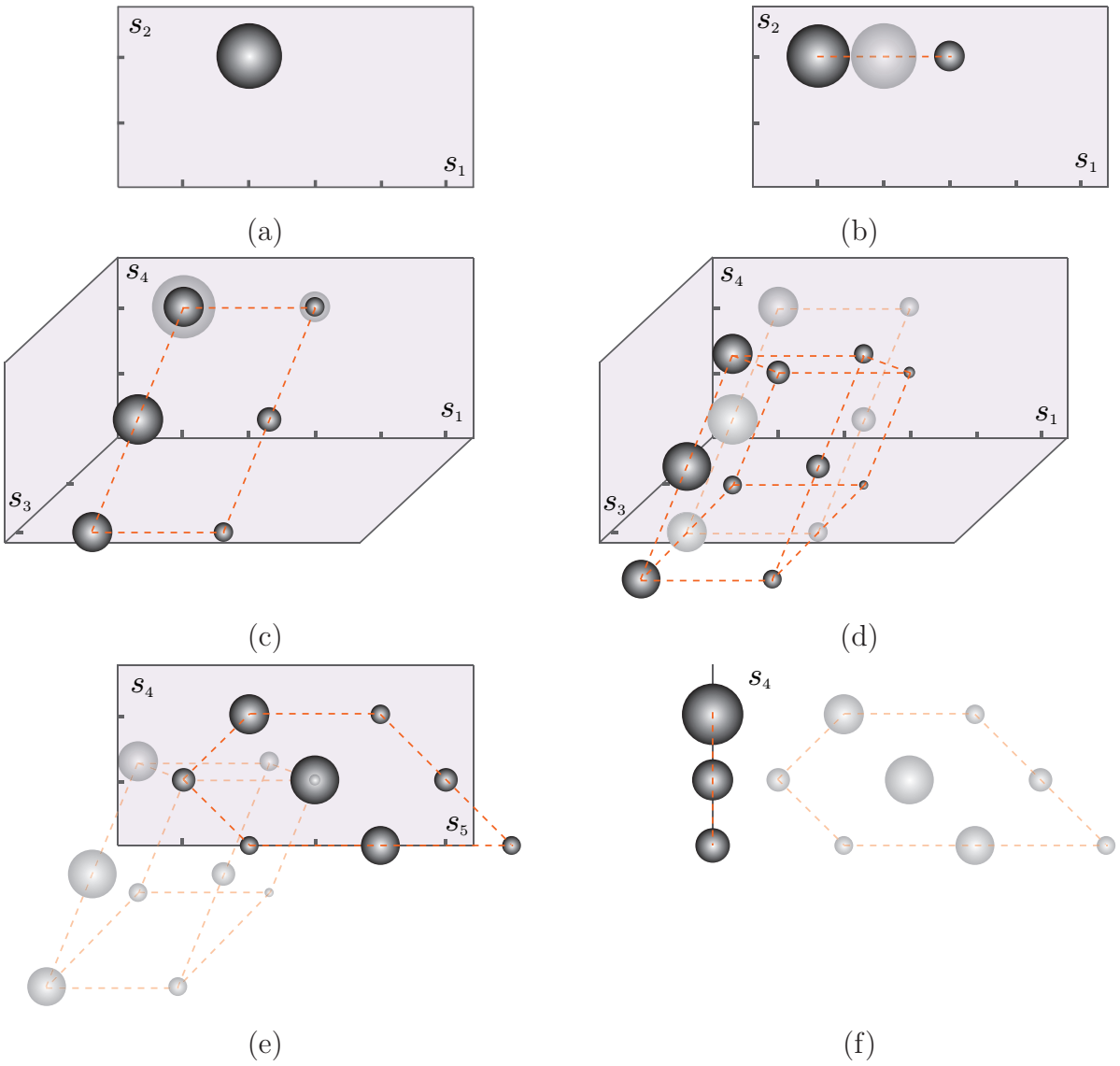


Figure 3.4: Graphical display of the target probability mass as we carry out the algorithm ProcessProbabilityMass over the simple event observation sequence in Fig. 3.3. Each figure corresponds to one step in Table 3.3. Lighter (if any) and darker balls represent probability masses before and after an event, respectively. The volumes are proportional to the magnitude of the probability mass entries.

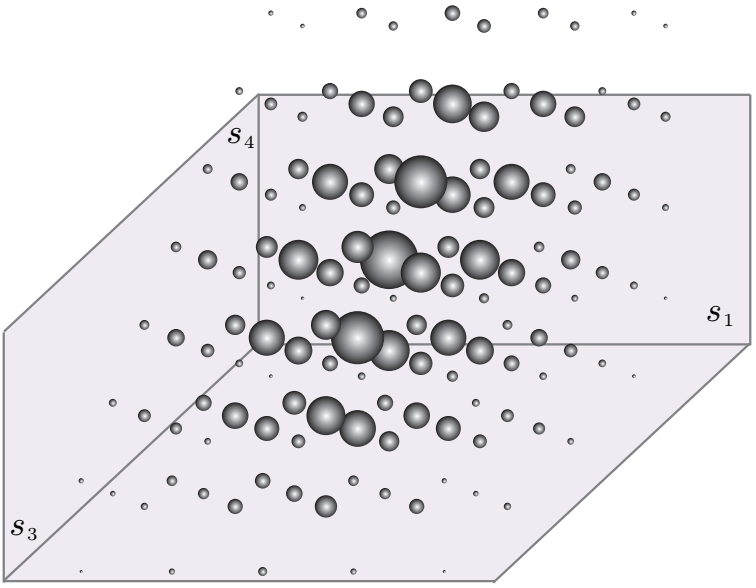


Figure 3.5: Probability masses for the more complicated observation sequence in Fig. 3.3 before shadows s_1, s_3 merge to form s_5. The axes s_1, s_3, and s_4, as shown in the figure, have ranges $[1, 9], [0, 7]$, and $[0, 5]$, respectively, starting from the origin.

tions by querying a Monte Carlo simulator. During each trial, the outcome of simulation may contradict an observation, in which case the trial is simply discarded. After a certain number of successful trials are completed, the final target distribution is obtained. For example, the mean of the number of targets in a shadow at $t = t_f$ is simply the average of the number of targets in that shadow over all successful runs. For simulations in this chapter, we require 1000 successful trials. Note that since the particular Monte Carlo simulation we perform in this chapter does not depend on data, its result is probabilistically correct and therefore can serve as baselines for verifying results from other algorithms.

Improving the ProcessProbabilityMass algorithm

Observing that the computation is burdened by storing the sheer amount of probability mass entries when there are many targets and observations, an obvious simplification is to *resample* the entries and keep the important ones. For example, we may choose to retain the first 1000 probability mass entries of largest value. With each step of processing looking

at each entry once, the processing time per step becomes a constant, albeit a large one. With this approximation, the earlier algorithm then runs in time *linear* in the number of observations. We call this heuristic *basic truncation*.

The problem with basic truncation, however, is that the trimmed away entries may turn out to be important. Take the processing in Table 3.3 for example, if the fourth entry after the merge step, $P(s_4 = 0; s_5 = 2) = 0.0225$, is truncated, then the second entry in the end, $P(s_4 = 0) = 0.0769$, will be lost, which is significant. The issue becomes problematic very quickly as the number of coexisting shadows increases, since each shadow creates one dimension in the joint distribution and sampling a high dimensional space is inherently inefficient. To alleviate this problem, in addition to the basic truncation approach of keeping fixed amount of entries with highest probability after each update, we also employ the following:

1. Randomly allow probability mass entries with low value to survive truncation. In doing this, we hope to allow enough low probability yet important entries to survive truncation. We denote this heuristic as *random truncation*.

2. Retain more entries during update steps right before merge and disappear events. Since disappear events usually cause the the number of probability mass entries to decrease dramatically (concentrating the distribution, or reducing the uncertainty), we can afford to keep more entries right before these events, without incurring much extra computational cost. Merge events also cause the number to decrease as some entries can be combined after merging. We combine this with random truncation and denote the resulting heuristic *random truncation with event lookahead*.

By construction, it is straightforward to see that the additional heuristics do not need asymptotically more time. There is a clear similarity between these heuristics and particle filtering: They all begin with a discrete set of probability masses, push the set through an update rule, and resample when necessary. They also share the same weakness: If the key sample

points with low probabilities are truncated, the end result may be severely skewed. Unlike in typical particle filter problems, the number of random variables in our problem keeps changing with split and merge events.

3.2.7 Extensions

In Subsection 3.2.1, assumptions (1) and (2) are made to simplify the presentation of the probabilistic algorithms. The first assumption is that component events are observed without error. Although component events can be observed with high accuracy in many environments, it is not always the case. Sensor network is such an example: Sensing range can be hard to know precisely. The extension to handle such uncertainties is relatively straightforward, at least in theory. All we need to do is to maintain a probability distribution over all possible sequence of shadows consistent with the robot's observations. Obtaining expectations of the number of targets in any shadow can then be done by also calculating the expectation over all possible sequences of shadows that contains the target shadow. The computation effort will certainly increase; resampling can alleviate the burden somewhat.

Various distinguishability assumptions can also be handled. Recall that in the nondeterministic formulation, two distinguishability cases are investigated. When there are only teams with single attributes, the approach from the nondeterministic setting applies by simply carrying out one computation per team. If the teams have multiple attributes (for example, the initial condition may be given as a joint distribution of red and blue teams), a direct extension is performing one computation for each joint probability entries in the initial condition. This is clearly more work and resampling may be necessary depending on the granularity of the initial target distribution. On the plus side, although we lose some accuracy with resampling (to save computation time), a richer class of problems can now be handled because any initial condition can be described as a joint probability distribution.

For the probabilistic case, we ran a simulation with the observation sequence in Fig. 3.6.

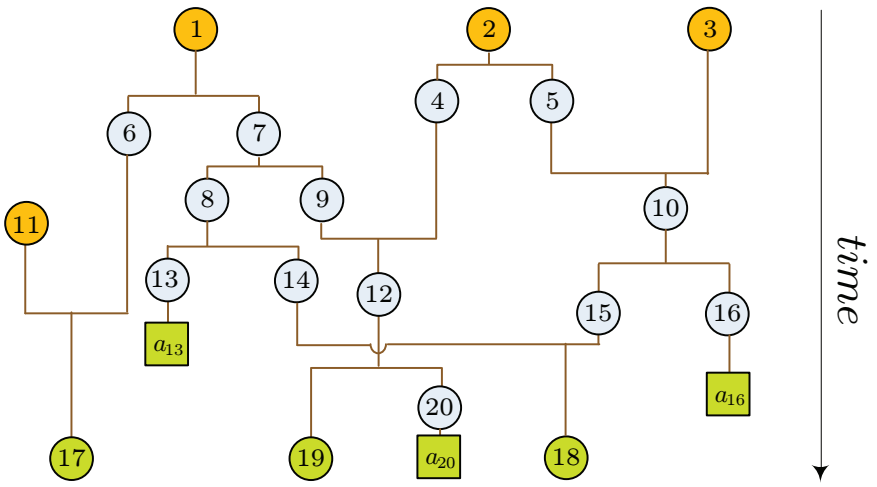


Figure 3.6: An event observation sequence with a total of 20 shadows in its life cycle. The FOV observations are not marked.

The sequence contains 14 component event observations. We also included 32 FOV events scattered along the sequence, which are not marked in the figure. Shadows $1, 2, 3, 11, 13, 16, 20$ are associated with $10, 7, 8, 9, 6, 9, 4$ targets (with probability 1), respectively. For performance measures, we look at the time for one run of the algorithm to complete, as well as the expectation (mean and standard deviation for randomized methods) of targets in individual shadows at the end (s_{17}, s_{18}, and s_{19}). When we randomly pick entries to keep, the time result is averaged over 10 runs and the accuracy is given in the form of mean and standard deviation. In our implementation, we also make the following choices: (1) for random truncation, the entries are kept based on its probability multiplied with a random number in $(0, 1)$, and (2) for event lookahead, we will not truncate the entries if there is a disappear event within the next four events or a merge event within the next two events. The outcome is summarized in Table 3.4. The heuristics basic truncation, random truncation, and random truncation with event lookahead are shortened as TR, RT, and RT-LA, respectively. The number following the method is the number of entries kept. By *frequent failure*, we mean that more than one-third of the times the heuristic fails to give a valid result. These are indicative of minimum number of entries needed for the method to work.

Table 3.4: Simulation results for various probabilistic methods.

heuristic	s_{17}	s_{18}	s_{19}	t(s)
none, precise	11.12	5.84	5.60	329.5
TR-10000	failure			
TR-20000	10.67	4.85	5.33	6.1
TR-50000	11.00	5.38	5.52	15.5
TR-100000	10.96	5.59	5.53	29.4
TR-200000	11.06	5.73	5.56	61.4
RT-10000	frequent failure			
RT-20000	11.38(0.20)	5.31(0.23)	5.67(0.20)	6.1
RT-50000	11.16(0.02)	5.36(0.03)	5.64(0.02)	14.6
RT-100000	11.03(0.01)	5.62(0.01)	5.56(0.01)	28.3
RT-LA-2000	frequent failure			
RT-LA-5000	11.32(1.30)	6.54(1.46)	5.28(1.28)	2.0
RT-LA-10000	11.17(0.56)	5.87(0.91)	5.02(0.48)	4.2
RT-LA-20000	11.62(0.26)	5.30(0.18)	5.57(0.16)	8.3
RT-LA-50000	11.32(0.01)	5.51(0.01)	5.60(0.01)	17.7
Monte Carlo	11.16	5.58	5.57	42.1

The result shows that when no heuristic is used, the algorithm takes much more time to finish. This is not surprising since the time complexity is induced by the space requirement for storing the probability mass entries. On the other hand, all of the truncation heuristics work reasonably well, with the randomized truncation plus event lookahead greatly reduces the number of entries to retain. The RT-LA-50000 run compares well with the TR-100000 run on accuracy, but uses one-third less time. We expect the advantage to become more obvious as more targets are present in the system. The final target distribution from one RT-100000 run is given in Fig. 3.7.

For a second test, we change the number of targets in shadows $1, 2, 3, 13, 16, 20$ to $25, 22, 23, 8, 15, 9$, while leaving other observations unchanged. With the increased number of targets, the basic algorithm runs out of memory after 10 minutes, before the third split is completed. At the peak of its memory usage during the failed run, there are more than 2×10^7 probability mass entries. On the other hand, the randomized methods do not have this problem. Both RT-100000 RT-LA-50000 yield good results compared to Monte Carlo

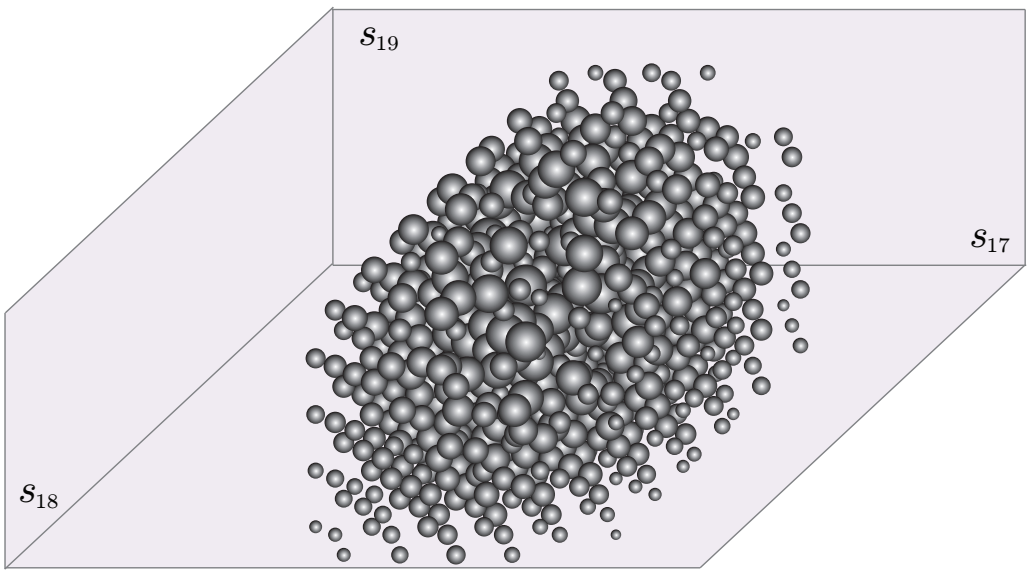


Figure 3.7: Probability masses after a RT-100000 run. The axes s_{17}, s_{18}, and s_{19}, as shown in the figure, have ranges $[12, 30], [12, 28]$, and $[7, 15]$, respectively, starting from the origin. The origin is shifted for better display.

trials, with similar running time. The result is summarized in Table 3.5. Comprehensive

Table 3.5: Simulation results on various probabilistic methods with over a hundred targets.

Heuristic	s_{17}	s_{18}	s_{19}	t(s)
none, exact	out of memory after 10 mins			
Monte Carlo	18.26	22.25	11.85	43.2
RT-100000	17.78(0.03)	21.83(0.03)	12.34(0.02)	66.3
RT-LA-50000	18.24(0.08)	21.99(0.07)	12.57(0.07)	40.6

performance analysis of probabilistic algorithm is hard since the performance depends on external factors such as the implementation of the specific split rule, random number generator, and so on. Nevertheless, for completeness, we discuss the performance at a higher level. To avoid the issue of external factors, we assume that at each step, each probability mass takes constant time to process. Unlike the nondeterministic case, running time of the ProcessProbabilityMass algorithm may depend heavily on the number of targets in the system via the split rule. If there are n split events with an average number of targets in the originating shadow being p and also n_f FOV events, with the reasonable additional

assumption that merge, appear, and disappear events are on the same order as split events, the ProcessProbabilityMass algorithm can take time $O(p^n 3^{n_f})$. The running time of the resampling based algorithms has a big constant depending on the number of entries to keep, but otherwise depends only linearly on the number of critical events.

CHAPTER 4

PATH INFERENCE VIA SENSOR FUSION

4.1 Sensor Based System Validation

One night, a crime was committed in an office building with complex interior structure. The next morning, a few suspects were identified but none of them would come forward. Instead, all of them provided seemingly convincing stories that excused them from being present at the crime scene. Unknown to the suspects, however, the building's security system, composed of a set of sensors with different capabilities, had made a sequence of recordings of passing people. Knowing that the criminal among the suspects was lying, can we use the sensor recordings to help solve the crime?

Similarly, in computer science, robotics, and control, a frequently encountered problem is verifying that an autonomous system, be it a program or a robot, is performing as designed. For example, a service robot may plan a path to clean office rooms one by one. Due to internal (sensor/actuator/computing units malfunctioning) or external factors (strong electromagnetic interference, for example), the robot may mistake one room for another and fail to accomplish its task without knowing that it has failed. A robot or a system may also be compromised for malicious purposes, producing intentionally bogus records of its actual path to hide the fact. In such cases, it would be highly desirable if external monitoring could automatically determine that a robot has faltered.

Switching to another perspective, due to diminished cost, an ever-increasing number of heterogeneous, more reliable sensors are being installed and many are also networked. Some

examples are: (1) surveillance cameras at supermarkets, (2) occupancy sensors in office for light control and homes for intrusion detection, and (3) buried pressure based vehicle sensors at street crossings. Sensors are everywhere. Some of these sensors are part of smart systems, such as automatic traffic citation issuing systems using cameras installed at crossroads. However, this vast network of sensors is far from realizing its full potential since the integration of sensor data usually requires human intervention. For example, video surveillance systems often merely fuse images from different cameras for human interpretation, which can be error prone. More effective use of networked sensors then calls for the development of specialized and efficient algorithms for these systems.

In this chapter, we introduce realistic abstractions of the afore mentioned problems and show that such formulations are computationally tractable. Specifically, one or more agents (robots or people) are assumed to move in an indoor environment, of which regions are monitored by external sensors (beam detectors and occupancy sensors). We assume that the agents are not aware of these sensors. From a story provided by an agent, which is a sequence of places in the environment it has visited, and combined recordings of these sensors, we provide polynomial time algorithms (with respect to the complexity of the environment, the length of the story, as well as the length of the observation history) for the inference problem of whether the given story is consistent with the sensor recordings. In presenting the algorithm, we demonstrate that this problem can again be viewed as a combinatorial filtering problem: We may filter the story through the sensor recordings and vise versa. Here, interestingly, the choice of the filtering order affects the efficiency of the resulting algorithm. After establishing the base solution for a single agent with exact story from the agent, scenarios involving multiple agents and approximate story validation are discussed in detail. Again, we provide efficient algorithms for solving most of these extensions.

Our work takes inspirations from two active research topics in robotics and control. If one assumes that the behavior of a set of moving bodies is largely unknown, the verification

problem becomes inferring various properties of these moving bodies with a network of simple sensors. Binary proximity sensors have been employed to estimate positions and velocities of a moving body using particle filters [55] and moving averages [56]. The performance limits of a binary proximity sensor network in tracking a single target are discussed and approached in [57], followed by an extension to the tracking of multiple targets [58]. The task of counting multiple targets is also studied under different assumptions [59, 60]. In these works, the sensor network's aggregate sensing range must cover the targets of interest at all times, which is much more difficult to implement than guarding critical regions of an environment. When only subsets of an environment are guarded, *word problems in groups* [61, 62] naturally arise. For the setup in which targets moving inside a 2D region are monitored with a set of detection beams, [63] characterizes possible target locations, target path reconstruction up to homotopy, and path winding numbers. In this domain, the surfacing of more interesting behaviors also induces an increase in complexity; few efficient algorithms exist. This prompts us to ponder: Can we do better if partial knowledge of a target's behavior is available? In viewing its resemblance to the questions asked in [55, 57, 63], our problem requires the design of a combinatorial filter, similar to those in [96, 87, 71]. These combinatorial filters are minimalist counterparts to widely known Bayesian filters [24, 25, 26, 27, 28, 10, 72].

On the other hand, if sensors external to moving bodies are ignored, one is left with the task of systematically verifying that the moving bodies do not have unexpected behaviors. Complex moving bodies such as robots are often modeled as hybrid systems. Existing verification techniques either address subclasses of hybrid systems or approximate reachable sets of such systems [64, 65, 66], because the problem of verifying a system with continuous state space and control input is generally undecidable [67]. In practice, this difficulty translates into the necessity of external measures to safeguard the unverified portion of a system. Alternatively, when high level task specifications can be coded as General Reactivity(1) formulas [68], the task of composing controllers into verifiably correct hybrid automata can be

carried out automatically using linear temporal logic [69, 70]. Even for such provably correct designs, malfunction can still occur due to sensor/actuator/computer errors. Keeping these systems in check again requires monitoring with external sensors.

In obtaining solutions to these path inference problems, it becomes clear that the problems we raise can be transformed to the *string edit distance problem between a string and a regular language*, with the first algorithmic solution appearing in [97]. Improvements on time and space requirements for algorithms that solve such problems can be found in [98, 99, 100, 101]. The best general algorithm for obtaining a string with the smallest edit distance from a string x between x and an automaton A appears to be the one given in [101], which takes time $O(|x||A| \lg |A|_Q)$, in which $|A|$ is the sum of the number of states/transitions of A, and $|A|_Q$ is the number of states of A. For an overview of approximate string matching problems, see [102].

The work presented in this chapter is based on [103, 104] and contributes to the research in robotics in two aspects. First, using a sparse network of simple sensors to validate the claimed behavior of an autonomous agent introduces a new methodology that complements traditional system verification techniques, such as those from [64, 65, 66]. We believe this is a necessary approach given that most verification processes focus on high level abstractions of an autonomous system, which only models simplified, ideal behavior. Second, applying principles of dynamic programming [105], we show that polynomial time algorithms exist for the proposed decision problems, providing insights into the structure of these detective game like problems. Moreover, the practical algorithmic solution may readily find its way in real-world applications, such as system design/monitoring/verification, security, and sensor-based forensics.

The rest of this chapter is organized as follows. Section 4.2 presents the problems on exact path inference and perform basic analysis of the structures these problems. Section 4.3 then continues to provide the algorithms for these problems and the their time complexities, as

well as connecting the solutions to combinatorial filters. In Section 4.4, these results are extended to approximate path inference, in which much more general problems are tackled.

4.2 The Exact Path Inference Problem

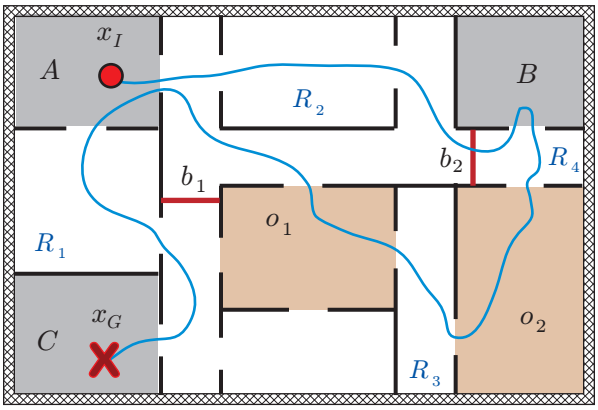


Figure 4.1: A workspace with rooms A, B, C, occupancy sensors o_1, o_2 and beam detectors b_1, b_2. The curve connecting the start (circle) and goal (cross) locations is a possible agent path for the story A, B, A, C, which triggers the sensor recordings b_2, o_2, o_2, o_1, o_1, in that order.

4.2.1 Workspace, agent stories, and observation history

Let the *workspace* $W \subset \mathbb{R}^2$ be a bounded, path connected open set with a polygonal boundary, ∂W. Let one or more point agents move around in W, carrying out unknown tasks. Every agent has a map of W and may move arbitrarily fast along some continuous path $\tau : [t_0\ ,\ t_f] \to W$. We are interested in a particular agent x which can be thought of as a *suspect*. Agent x provides a *story*, which is a sequence of locations it recalls along its path in increasing chronological order,

$$\mathbf{p} = (p_1, p_2, \ldots, p_n), \quad p_i \subset W.$$

Since the p_i's can be viewed as letters from an alphabet, $\mathbf{p}$ can be viewed as a string $p_1 \dots p_n$ as well; we use the string notation mostly since it is shorter. We assume that the unique elements of $\mathbf{p}$ are each a simply connected region with a polygonal boundary and pairwise disjoint. The set of all unique elements of $\mathbf{p}$ is denoted $\mathcal{C}_p$, which can be thought of as a set of rooms in W. As an example, for the environment from Fig. 4.1, agent x may simply report that "I went from room A to room B, then came back to room A, and eventually arrived room C, at which point I stopped." This translates the story to $\mathbf{p} = (A, B, A, C)$ or simply $\mathbf{p} = ABAC$. To limit the number of questions that can be posed, we require that agent x starts from p_1 and ends in p_n. Unless otherwise stated in a particular problem, it is assumed that agent x recalls in $\mathbf{p}$ locations on its path correctly.

Let a subset of the workspace W be guarded by a set of occupancy sensors and beam detectors. The placement of sensors in W is unknown to the agents. An occupancy sensor o_i is assumed to detect the presence of an agent in a fixed, convex subset $c \subset W$. For example, a room may be monitored by such a sensor (o_1, o_2 in Fig. 4.1). A data point recorded by o_i has two parts, an *activation*, $s_{oa} = (o_i, t_a)$, and a *deactivation*, $s_{od} = (o_i, t_d)$. Here s_a is the time when the first agent enters an empty c and t_d is the time when the last agent exits c. A beam detector b_i, on the other hand, guards a straight line segment, $\ell \subset W$, between two edges of ∂W (b_1, b_2 in Fig. 4.1). A data point of such a sensor is recorded as an agent crosses ℓ, which can be represented by a 2-tuple, $s_b = (b_i, t)$. A beam detector is deactivated right after activation. We further assume that when a beam detector is triggered by an agent, the agent must pass from one side of the beam to the other side.

If we gather all sensor recordings (of the types s_{oa}, s_{od}, s_b) during a time interval $[t_0, t_f]$ and order them by time incrementally, an *observation history* is obtained. This sequence is unique since it is reasonable to assume that no two recordings happen at the exact same time. As we do not assume that an agent provides the exact time when it visits a location, the time in the sensor recording is also relative. Therefore, when we compose the observation history

of the sensors, we may discard the time element of each sensor recording, keeping only their relative order. A simplified observation history can be written as (each s_i corresponds to the region covered by a sensor):

$$\mathbf{s} = (s_1, s_2, \ldots, s_m), \quad s_j \subset W.$$

Similar to $\mathbf{p}$, we can write $\mathbf{s}$ as a string. Note that m, n are of comparable size since the more places an agent visits, the more sensor recordings it triggers; the relationship between m, n is roughly linear. Also, for an occupancy sensor, there should always be an even number of appearances of it in $\mathbf{s}$, with the odd numbered appearances being activation and the even numbered appearances being deactivation. If there is a single agent in the workspace, an activation must also be followed by a deactivation of the same occupancy sensor. For example, if there is a single agent with $\mathbf{s} = (o_1, o_1, b_2)$ or simply $\mathbf{s} = o_1 o_1 b_2$, then the agents first enters the region guarded by o_1, then exits o_1, and passes through b_2.

In the example given in Fig. 4.1, a sensor recording of an occupancy sensor could imply that the agent enters and exits from any of the doorways; there are four doorways for o_1 and two for o_2. Similarly, when the beam detector b_2 is triggered, an agent could be passing it from left to right or in the other direction. These sensors certainly cannot distinguish among different agents. Since we work with these weak sensors, the algorithms in this chapter apply to a wider range of sensors, provided that they are at least as powerful. We denote the unique sensor regions from $\mathbf{s}$ as $\mathcal{C}_s$.

4.2.2 Sensor based story validation and path inference

Given the setup from Subsection 4.2.1, we want to know whether a story is consistent with a sensor observation history. For example, if an agent starts from A in the workspace from Fig. 4.1, it cannot end up at B without triggering any sensor recordings. That is, sensor

recordings limit the possible paths an agent may take in a given environment and therefore restrict the possible stories an agent may provide. To investigate questions of this flavor, we first look at the problem of validating a single agent's story against the sensor observation history recorded during the same time interval, in the presence of no other agents in the same workspace. We also assume that the agent's story is exact: The suspect agent x cannot have error in its description of a story. We formulate the problem of verifying an agent's story in a single-agent workspace or a multiple-agent workspace as follow.

Problem 1 (Single-agent workspace, exact story, matching time intervals) *Let there be a single agent in a workspace. The agent provides a story* $\mathbf{p}$ *for the time interval* $[t_0, t_f]$*. During the same time interval, the sensors in the workspace produce an observation history,* $\mathbf{s}$*, of the agent's activities. Validate whether* $\mathbf{p}$ *is consistent with* $\mathbf{s}$*. Extract a feasible path if the story is consistent.*

Problem 2 (Multiple-agent workspace, exact story, matching time intervals) *Let there be an unknown number of agents in a workspace. One of the agents, say agent* x*, provides a story* $\mathbf{p}$ *for the time interval* $[t_0, t_f]$*. During the same time interval, the sensors in the workspace produce an observation history,* $\mathbf{s}$*, of all agents' activities. Validate whether* $\mathbf{p}$ *is consistent with* $\mathbf{s}$*. Extract a feasible path if the story is consistent.*

As an example, again using the environment from Fig. 4.1, agent x's story may be

$$\mathbf{p} = ACBAC. \tag{4.1}$$

In English, agent x told the story that it started in room A and went through room C, B, A, and C, in that order. When there is a single agent in the workspace, we let the observation history be

$$\mathbf{s} = b_1 o_1 o_1 b_2 o_2 o_2. \tag{4.2}$$

For this simple example, it is not hard to see that $\mathbf{p}$ is not consistent with $\mathbf{s}$: B can only be visited after agent x passes b_2 from left to right; however, after visiting B, either b_2 or o_2, o_1 must be triggered for x to visit A once more. On the other hand, if instead we have the same story $\mathbf{p}$ but the following observation history,

$$\mathbf{s}' = o_1 o_1 o_2 o_2 b_2, \tag{4.3}$$

then the story is consistent with sensor observation. One possible trajectory the agent may take to pass through the rooms and sensors regions is $ACo_1o_1o_2o_2Bb_2AC$, which is shown as the blue curve in Fig. 4.1. Note that an uncountable number of such consistent trajectories may be possible, forming a countable number of homotopic path classes.

When there are multiple agents (say three agents) in the workspace, an observation history with which $\mathbf{p}$ is consistent is

$$\mathbf{s}'' = b_1 o_1 o_2 b_2 o_2 o_1. \tag{4.4}$$

A possible sequence of events for the agent of interest (say agents x) is $\mathbf{p} = Ab_1CBb_2AC$. The agent could cross o_1, o_2 without triggering any sensor recording if the other two agents enter o_1, o_2, respectively, before x does and leave o_1, o_2 later than agent x does. One possible set of trajectories for the agents are shown in Fig. 4.2. In the examples, we have implicitly made the assumption that elements of $\mathcal{C}_p$ and elements of $\mathcal{C}_s$ have coverage regions that are pairwise disjoint; overlapping cases will be handled after the main algorithms are introduced.

4.2.3 The connectivity graph

Both occupancy sensors and beam detectors, when not triggered, act as obstacles that change the workspace connectivity. When a sensor is triggered, the part of the workspace blocked by that sensor is temporarily connected. To explore the structure from this observation, we

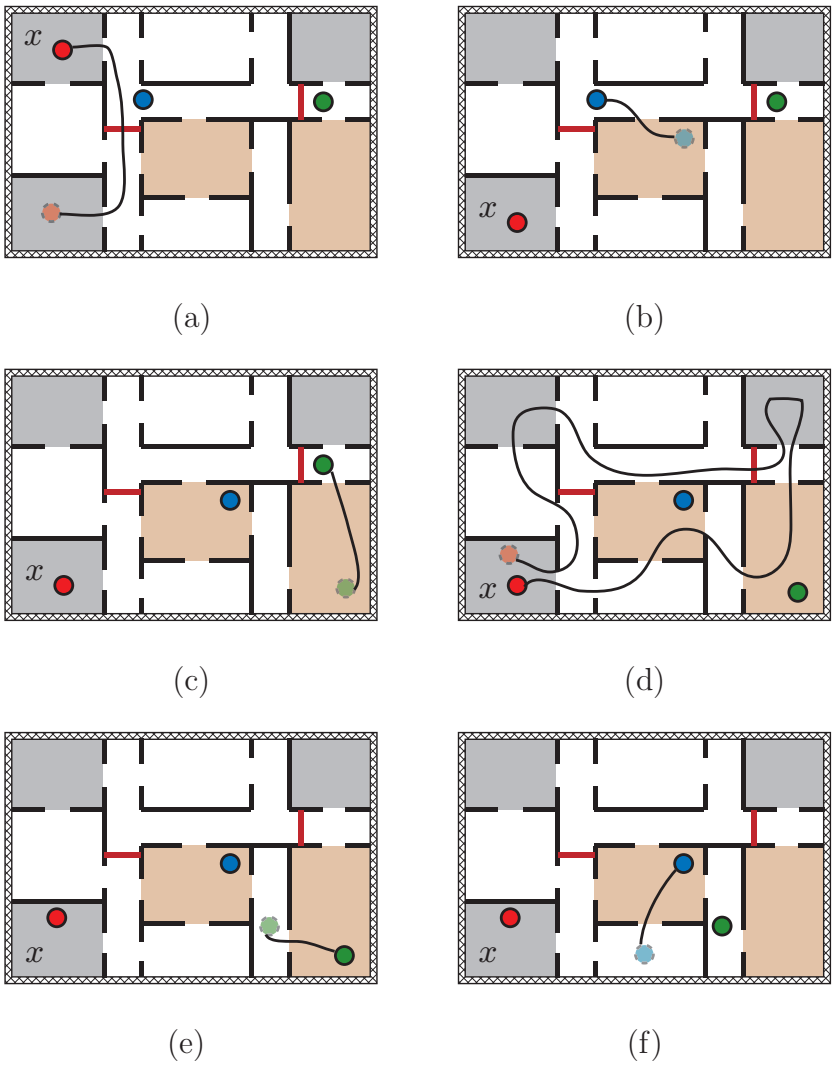


Figure 4.2: A possible set of trajectories for a multiple agents scenario. (a) Agent x (red) moves from room A to C, crossing b_1. (b) Blue agent moves into o_1 and stays there. (c) Green agent moves into o_2 and stays there. (d) Agent x moves through o_2, o_1, room B, b_2, A, and C, triggering only b_2 during the process. (e) Green agents moves out of o_2. (f) Blue agent moves out of o_1.

first build a *connectivity graph* G that captures the topological features of W. As we are only interested in finding a path[1] through $\mathbf{p}$ that is compliant with $\mathbf{s}$, we only need G to capture how elements of $\mathcal{C}_p$ are connected and how they are connected to the sensors, $\mathcal{C}_s$. Therefore, we treat these elements as vertices of G. Since there are two possible directions an agent may pass through a beam detector, two vertices are needed for each beam detector. A single vertex is needed for each element of $\mathcal{C}_p$ and for each location guarded by an occupancy sensor. For the example from Fig. 4.1, the collection of vertices is

$$V = \{A, B, C, o_1, o_2, b_{1u}, b_{1d}, b_{2l}, b_{2r}\},$$

in which b_{1u}, b_{1d} are the upper and lower sides of b_1, respectively (these two sides are naturally obtained if a beam is represented as two oppositely oriented edges, as commonly used in computations involving polygons). Similarly, b_{2l}, b_{2r} are the left and right sides of b_2.

Algorithm 3 BUILDCONNECTIVITYGRAPH

Input: W_{free}, V
Output: $G = (V, E)$, the connectivity graph of W_{free}

```
C ← VERTCELDECOMP(W_free)
C' ← COMBINENBRCELLS(C)
initialize E as an empty set
for each connected component c_i ∈ C' do
    initialize V_I as an empty set
    for each edge e_j of c_i do
        if e_j is an edge of some sensor v_k ∈ V then
            add v_k to V_I
        end if
    end for
    for each pair of v_j, v_k ∈ V_I do
        E ← E ∪ {(v_j, v_k)}    //   (v_j, v_k) denotes an edge.
    end for
end for
return (V, E)
```

[1] In graph theory, a path does not visit one vertex multiple times. Therefore, the image of a continuous path, when discretized, becomes a *walk* in graph theory terminologies, since it may visit a vertex multiple times. In this chapter, we abuse the term *path* slightly to denote both a continuous function $\tau : [t_0, t_f] \to W$ and the corresponding walk in a discrete graph.

To connect the vertices, we need to obtain the connectivity of the workspace algorithmically, treating the regions occupied by elements of $\mathcal{C}_p$ and $\mathcal{C}_s$ as obstacles. Denoting the workspace excluding these obstacles as W_{free}, determining the connectivity of W_{free} is equivalent to finding the connected components of W_{free}. One efficient way of doing this is to apply a cell decomposition procedure (see [3], Chapter 6), such as vertical cell decomposition, to W_{free} and then combine the cells that share borders. For each connected component of W_{free}, we can check whether it borders elements from V (viewing elements of V as regions instead of vertices). An edge is then added for any two elements of V that borders the same connected component of W_{free}. The pseudocode for extracting G is summarized in the routine BUILDCONNECTIVITYGRAPH (Algorithm 3) assuming that W_{free} is given as a list of oriented, simply connected polygons. VERTCELDECOMP does vertical cell decomposition on W_{free} and COMBINENBRCELLS groups neighboring cells into a single cell. These are standard subroutines [106]. Figure 4.3 shows what happens to the region R_1 of our example

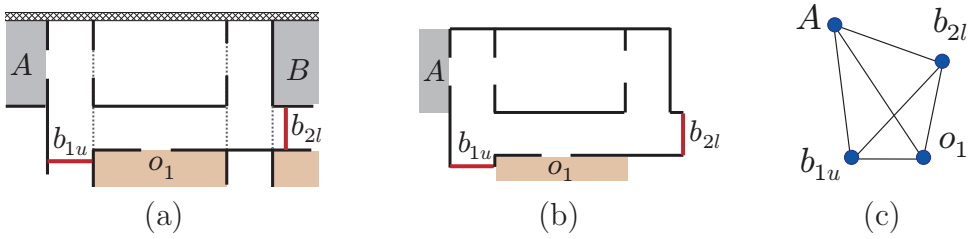


Figure 4.3: Region R_1 of Fig. 4.1 after cell decomposition (a), combining neighboring cells (b), and adding edges of G (c).

after BUILDCONNECTIVITYGRAPH is run. The complete connectivity graph $G = (V, E)$ is given in Fig. 4.4(a). We point out that there are other choices in constructing the connectivity graph. For example, following an (perhaps more natural) equivalence class approach, we may alternatively build the graph based on how regions R_1 through R_4 are connected (Fig. 4.4(b)). We may further treat sensors and rooms as directed edges. There are no fundamental differences between these choices for our purpose: Although the later choices provide simpler graphs, slightly more sophisticated graph search routines would then be needed.

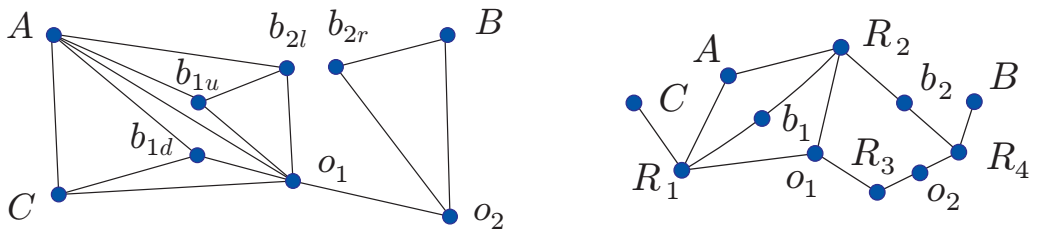


Figure 4.4: (a) The connectivity graph of the example given in Fig. 4.1. (b) An alternative connectivity graph including connected components of W_{free} as vertices.

Algorithm 4 GetSubGraph

Input: $G = (V, E)$, the start vertex v_s, $\mathcal{C}_p$, and goal vertices V_G
Output: $G' = (V', E')$, the part of G that is reachable from v_s

1: $V_C \leftarrow \mathcal{C}_p \cup \{v_s\}$
2: **return** GetReachableSubgraph(G, v_s, V_C, V_G)

4.2.4 Sensing induced subgraphs

With G constructed, we can now explore the extra information provided by the observation history: The relative timing of sensor recordings. This information essentially partitions G into different pieces at different time instances. In this section we focus on the case of workspace with a single agent. Assuming the observation history given in (4.2) which we may rewrite with time information as $\mathbf{s} = ((b_1, t_1), (o_1, t_2), (o_1, t_3), (b_2, t_4), (o_2, t_5), (o_2, t_6))$, b_1 is the first sensor that is set off. This means that at the time right before t_1 when the sensor is activated, the agent must be at either b_{1u} or b_{1d}. During the time interval $[t_0, t_1)$, since b_2, o_1, and o_2 are inactive, they act as obstacles. The part of G that the agent may travel during $[t_0, t_1)$ is then given by G_1 in Fig. 4.5, in which A is the start vertex and b_{1u}, b_{1d} are the possible goal vertices. Vertex B does not appear in G_1 because it is not reachable. Similarly, we obtain the subgraphs of G during time intervals $(t_1, t_2), (t_3, t_4), (t_4, t_5), (t_6, t_f]$ as G_2 through G_5 in Fig. 4.5, respectively. Graph G_2 has two parts since there are two possible start vertices. Note that when the start and goal vertices in these subgraph correspond to sensor vertices, they can be visited only once as the start vertex or the goal

Algorithm 5 GetReachableSubgraph

Input: $G = (V, E)$, v_s, V_C, V_G
Output: $G' = (V', E')$

```
1: for all edges (v_i, v_j) ∈ E such that v_i, v_j ∈ V_C do
2:    add (v_i, v_j) to E'    //   V' is also updated.
3: end for
4: G' ← ConnectedComponent(G', v_s)
5: if V_G is not empty then
6:    for all v_i, v_j such that v_i ∈ V', v_j ∈ V_G do
7:       if (v_i, v_j) ∈ E then
8:          add (v_i, v_j) to E'
9:       end if
10:   end for
11:   V' ← V' ∪ V_G
12: end if
13: return G'
```

vertex. The pseudocode is given in GetSubGraph (Algorithm 4). The algorithm calls the subroutine GetReachableSubgraph(G, v_s, V_C, V_G) (Algorithm 5), which returns the part of G reachable from v_s, passing only vertices in V_C. If V_G is not empty, then a path from v_s must also end at vertices of V_G. We separate this subroutine since it will be reused. In Algorithm 5, subroutine ConnectedComponent(G, v_s) returns the connected component of G containing v_s. We note that, although it is possible to work with G directly instead of working with these subgraphs, they will be helpful in understanding the algorithm and in complexity analysis. Moreover, it can be a good heuristic to build these subgraphs to restrict search in problems with large workspaces.

4.3 Efficient Algorithms for the Exact Path Inference Problem

4.3.1 Solving the single-agent problem

We are now ready to solve Problem 1. Having obtained the subgraphs of G, the rest of the work becomes searching through these graphs, one by one, for a path that agrees with

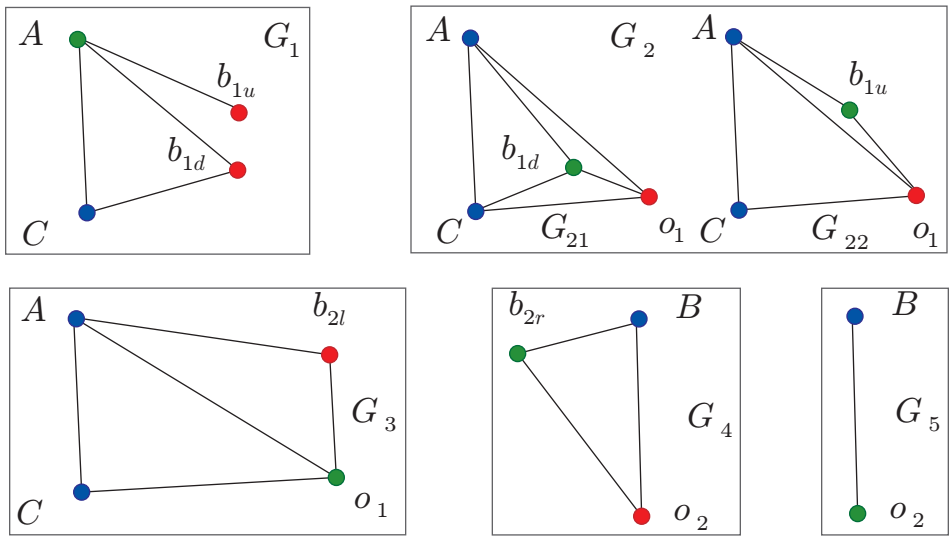


Figure 4.5: The subgraphs of G induced by the sensor observation history. The green vertices are possible start positions and the red vertices are possible goal positions.

the agent's story. A straightforward approach is to connect one subgraph's goal vertices to the next subgraph's start vertices and perform an exhaustive search through paths to see whether there are matches. Such naive algorithms are not scalable, however, since every beam detector can require connecting the subgraphs in two ways (for example, G_{21}, G_{22} in Fig. 4.5). The number of search paths through the subgraphs is then exponential in the number of sensor recordings on average. In the worst case, breadth-first or depth-first search through all these graphs may take an exponential amount of time.

To organize the search more efficiently, we first connect the subgraphs to get a better understanding of the topology of the graph to be searched. To make the structure more explicit for search, we also make the subgraphs directed. Doing this to all sensing induced subgraphs of G yields the graph in Fig. 4.6. Denote this graph G_s. The problem of validating $\mathbf{p}$ against $\mathbf{s}$ becomes searching through G_s for a path $\mathbf{p}'$ such that, after deleting the vertices corresponding to sensors from $\mathcal{C}_s$, $\mathbf{p}'$ is exactly $\mathbf{p}$. We observe that, since G_s contains at most $2(m+1)$ copies of G, any element of $\mathcal{C}_p \cup \mathcal{C}_s$ cannot appear more than $O(m)$ times in G_s. This observation indicates it may be possible to apply the principles of

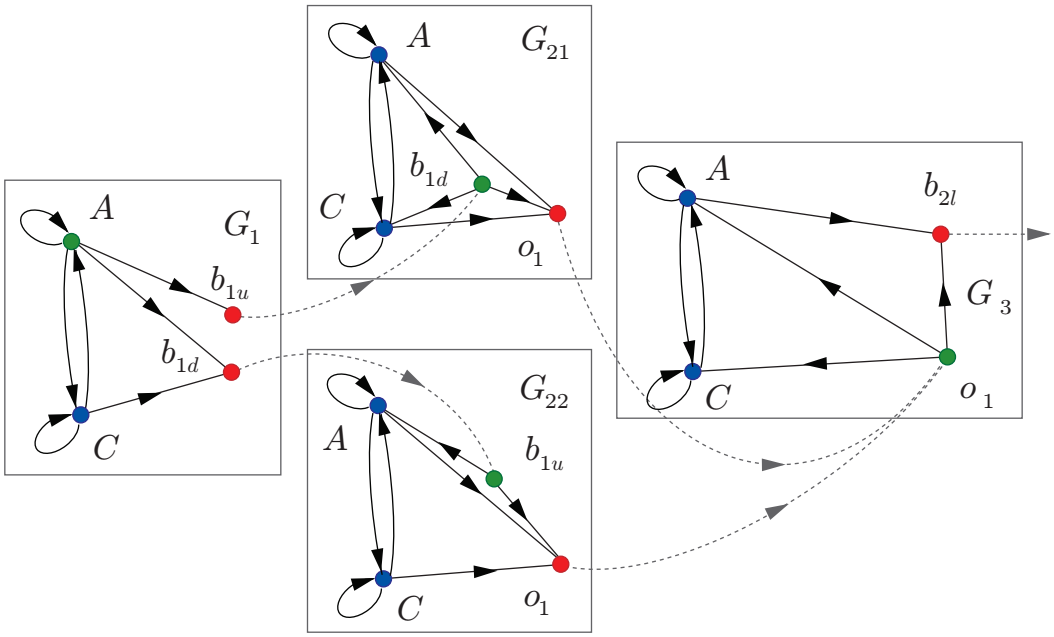


Figure 4.6: Part of the composite graph G_s built from the connectivity graph G and sensor observation history $\mathbf{r}$.

dynamic programming to partition the search problem into subproblems: Each subproblem is validating a tail $(p_i, \ldots, p_n)$ of $\mathbf{p}$, starting from a subset of vertices of G_s corresponding to p_{i-1}. The total number of subproblems per p_i is $O(m)$; if going from one subproblem to a smaller subproblem takes polynomial amount of time, then the total time spent on searching G_s is also polynomial.

This turns out to be the case for our problem. Before formally introducing the algorithm, we illustrate how it operates with the provided example. We write the agent's story compactly as $\mathbf{p} = ACBAC$. Since agent x starts in A, we are done with $p_1 = A$, leaving $CBAC$ to validate. For $p_2 = C$, it is possible to reach from A in G_1 to copies of C in G_1, G_{21} (passing b_{1u}, b_{1d}), and G_3 (passing b_{1d}, b_{1u}, o_1). The copy of C in G_{22} is not directly reachable from A in G_1, passing only vertices from sensors. We may write the three subproblems as (for $P \in \mathcal{C}_p$, P_{G_i} denotes that the copy of P is from the subgraph G_i. For example, A_{G_1}

denotes the copy of A from the subgraph G_1):

$$\begin{array}{r|l} A_{G_1}C_{G_1} & BAC, \\ A_{G_1}b_{1u}b_{1d}C_{G_{21}} & BAC, \\ A_{G_1}b_{1d}b_{1u}o_1C_{G_3} & BAC. \end{array}$$

Since there are multiple subproblems for $p_3 = B$, going through these subproblems individually may introduce a factor of $O(m)$ per problem; there can be $O(m)$ subproblems, which will contribute a factor of $O(m^2)$ to the overall running time. To avoid this, we again use the sequential nature of G_s. Instead of processing each subproblem individually, we process all of them together, staged at each G_j. For our example, the first subproblem starts with the copy of C in G_1: It is possible to go through b_{1d}, b_{1u} and get to o_1. We now pick up the second subproblem and see that it is possible to go from C in G_{21} to o_1 as well. At this point, the first two subproblems collapse into a single subproblem. Going into G_3, we pick up the third subproblem. For the copy of C in G_3, since it must pass A to reach B, this subproblem dies; we are left with a single subproblem to reach B from b_{21} in G_3. Following the given procedure, we obtain two subproblems after processing $p_3 = B$:

$$\begin{array}{r|l} A_{G_1}b_{1u}b_{1d}C_{G_{21}}o_1b_{21}b_{2r}B_{G_4} & AC, \\ A_{G_1}b_{1u}b_{1d}C_{G_{21}}o_1b_{21}b_{2r}o_2B_{G_5} & AC. \end{array}$$

Note that we do not keep all valid paths in this search; doing so will require space exponential in m. After all of $\mathbf{p}$ is processed, if some subproblems survive, then $\mathbf{p}$ is consistent with $\mathbf{s}$; any surviving subproblem also provides a feasible path.

4.3.2 The algorithm, its correctness and time complexity

The pseudocode is summarized in Algorithm 6. The subroutine IsDeactivation(r) returns true only if r is the deactivation of an occupancy sensor. The subroutine SensorVertices(r)

Algorithm 6 ValidateAgentStory

Input: G, $\mathbf{p} = (p_1, \ldots, p_n), \mathbf{s} = (s_1, \ldots, s_m)$
Output: **true** if $\mathbf{p}$ is consistent with $\mathbf{s}$, **false** otherwise

$V_I \leftarrow \{p_1\}$
for $j = 1$ to $m + 1$ **do**
  initialize V_G as an empty set
  if IsDeactivation(s_j) **then**
    continue
  end if
  if $(j \neq m + 1)$ **then**
    $V_G \leftarrow$ SensorVertices(s_i)
  else
    empty V_G
  end if
  $G_j \leftarrow$ GetSubGraph($G, V_I, \mathcal{C}_p, V_G$)
  $V_I \leftarrow V_G$
end for
$G_s \leftarrow$ Chain($G_1, \ldots, G_{m+1}$)
initialize V_s, V_s' as empty sets of two tuples
$V_s \leftarrow \{(p_1, 1)\}$ // A two tuple is a vertex of G_s
for $i = 2$ to n **do**
  for $j = 1$ to $m + 1$ **do**
    if (p_i, j) adjacent to $(p_{i-1}, k) \in V_s$ for some $k \leq j$ **then**
      if $i == n \&\& j == m + 1$ **then**
        return true
      end if
      add (p_i, j) to V_s'
    end if
  end for
  $V_s \leftarrow V_s'$; empty V_s'
end for
return false

returns the vertices of G induced by the sensor in a sensor recording r. The subroutine CHAIN(...) connects all input graphs sequentially based on sensor crossings, which is trivial to implement. In the code, we use (p_i, j) to denote the the copy of p_i in subgraph G_j.

As mentioned in the problem formulation, we have assumed that $\mathcal{C}_p$ and $\mathcal{C}_s$ do not overlap in $\mathbb{R}^2$. These are not included in Algorithm VALIDATEAGENTSTORY to avoid complicating the presentation. What if some $p \in \mathcal{C}_p$ and $s \in \mathcal{C}_s$ do overlap? There are several subcases. If the regions of p, s coincide (for example, there may be an occupancy sensor in room A), this essentially breaks the problem into several smaller problems, to which Algorithm VALIDATEAGENTSTORY applies. If $s \subsetneq p$, agent x then must go through p to reach s, in which case we can build G to make s a vertex connecting to p only. The same applies if $p \subsetneq s$. In the last case s, p partially overlap but do not include each other; we can treat s, p as three regions: $s \backslash p$ as a sensor, $p \backslash s$ as a room, and $s \cap p$ as a fully overlapping sensor and room (this case only happens to occupancy sensors, not beam detectors). This will split the verification problem into several subproblems, which may induce exponential growth in running time. However, the last case is not likely to often happen since occupancy sensors are usually placed to guard an entire room. We can also minimize such a problem by carefully placing the sensors.

The correctness of Algorithms 3 through 5 is by construction, which is relatively straightforward to verify. We now give an estimate of the worst case performance of these algorithms. Let W_{free} have an input size of n_w, VERTCELDECOMP can be computed in $O(n_w \lg n_w)$ time [106]; COMBINENBRCELLS takes time linear in n_w. For $G = (V, E)$, as an over estimate, we have $|\mathcal{C}_p|, |\mathcal{C}_s|, |V| \sim O(n_w), |E| \sim O(n_w^2)$. The three loops at lines $4, 6, 7$ in BUILDCONNECTIVITYGRAPH take no more than $O(n_w^2)$ time via amortization; the same is true for the loops at lines $4, 11$. The time complexity of BUILDCONNECTIVITYGRAPH is then no worse than $O(n_w^2)$. In the subroutine GETREACHABLESUBGRAPH, CONNECTEDCOMPONENT takes time linear in n_w [107]. The complexity is then decided by the "for" loop at line 6 and

the membership check at line 7, which takes no more than $O(|V_G| n_w \lg n_w)$ in total. Since $|V_G| \leq 2$ in calls to GetSubGraph, this routine takes time no more than $O(n_w \lg n_w)$.

The correctness of ValidateAgentStory follows from its construction based on dynamic programming, which we briefly corroborate. After each p_i is worked on, there are up to $O(m)$ subproblems since there are no more than $O(m)$ copies of p_i in G_s. Because the further observation that G_s is sequential, the suproblems for each p_i can be processed in a single pass of G_s. We make $O(m)$ calls to GetSubGraph, which takes $O(mn_w \lg n_w)$ total time. Going through the "for" loops, it is straightforward to get that the rest of the algorithm has complexity no worse than $O(n \cdot m \lg n_w) = O(nm \lg n_w)$. The worst case running time is then upper bounded by $O(m(n + n_w) \lg n_w)$.

4.3.3 The combinatorial filtering perspective

Before ending this section, let us inspect the proposed solution through the information space and combinatorial filtering goggle. As indicated by the proposed algorithm, we obtain a history information space by viewing a story as the actuation history and ambient sensors' recordings as the sensing history. The task of determining whether a story is consistent with the sensor recordings becomes validation whether the history information state is reachable or not in the history information space. By converting each sensor observation into a subgraph, we essentially created filters that are used to prune possible agent trajectories. Alternatively, we observe that it is also possible to build filters based on the elements of the agent's story. These filters can then be used to filter out sensor recordings that are impossible.

However, also as indicated by the preceding discussion, such direct applications of combinatorial filtering may not yield the most efficient algorithm. The reason for that is the structure of the information does not match that of a standard information space such as one depicted in Fig. 4.7 (the same as Fig. 1.4, which is reproduced here for convenience). The difference lies with the relative timing of the sensor recordings and agent actions: Such

$$
\begin{array}{lllll}
I_{hist} & \cdots \rightarrow & \eta_{t-1} & \xrightarrow{u_{t-1},y_t} & \eta_t \quad \rightarrow \cdots \\
\downarrow & & \downarrow & & \downarrow \\
I_{der} & \cdots \rightarrow & \eta'_{t-1} & \xrightarrow{u_{t-1},y_t} & \eta'_t \quad \rightarrow \cdots \\
\downarrow & & \downarrow & & \downarrow \\
\cdots & \cdots \rightarrow & \cdots & \longrightarrow & \cdots \quad \rightarrow \cdots
\end{array}
$$

Figure 4.7: Although it is possible to obtain $\eta'_t \in \mathcal{I}_{der}$ from $\eta_t \in \mathcal{I}_{hist}$, it is also possible to derive it from η'_{t-1} and $u_{t-1,t}, y_{t-1,t}$. The diagram commutes.

information is not available to us in the current problem. Therefore, at each step of the filtering process, the filter may need to process all of $\mathbf{p}$ (if we build filters based on sensor recordings), which is clearly inefficient.

Nevertheless, a basic analysis based on the information space framework is quite beneficial for two reasons. First, if we pick either the story or the sensor recordings to build filters, we do get reasonably efficient algorithms (the asymptotic time complexity will get bumped up by m or n, respectively. We leave this to the reader to verify). It may be the case that we cannot do any better. Second, the combinatorial filtering approach directly gives us two possible filter choices and their relationship in the filtering process, which provides quick guidance towards better filter design.

4.3.4 Solving the multiple-agent problem

We now move to Problem 2. When there are two or more agents in the workspace, two complications arise. First, as mentioned in Section 4.2.1, when multiple agents are in the workspace, there can be many agents in the region monitored by an occupancy sensor during one of its activation-deactivation time interval. Effectively, this allows any agent to temporarily go through the region monitored by an activated occupancy sensor. This suggests that the recordings from occupancy sensors should only be treated as events that change the connectivity of the environment. That is, whether agent x is the agent that triggered the activation/deactivation of an occupancy sensor is not relevant. Second, agent x may not

be responsible for all beam detector recordings. Instead, it may trigger any subsequence of sensor recordings. For an observation history sequence of length m, there are up to $O(2^m)$ possible subsequences; agent x may have triggered any one of these subsequences, but not the rest of $\mathbf{s}$.

To overcome these difficulties, we start by examining how the composite graph G_s changes. Given the analysis in the previous paragraph, only beam detector events need to be considered for agent x. For occupancy sensors, we maintain an active list as $\mathbf{s}$ is processed; additional processing is needed only when deactivation of an occupancy sensor happens. To illustrate the procedure for building the composite graph, we use $\mathbf{p}$ from (4.1) and $\mathbf{s}$ from (4.4), which we may rewrite as $\mathbf{s} = ((b_1, t_1), (o_1, t_2), (o_2, t_3), (b_2, t_4), (o_2, t_5), (o_1, t_6))$. Starting with the first beam detector recording, (b_1, t_1), if agent x is responsible for it, then the reachable part before b_1 is crossed is the same as G_1 from Fig. 4.5. To emphasize that this graph is built from t_0 to t_1, we denote it as G_1^0. The next two recordings in $\mathbf{s}$ are activations of o_1 and o_2. Since there are only activations, which only cause more locations of the environment to become reachable, we store these in the active occupancy sensor list and continue.

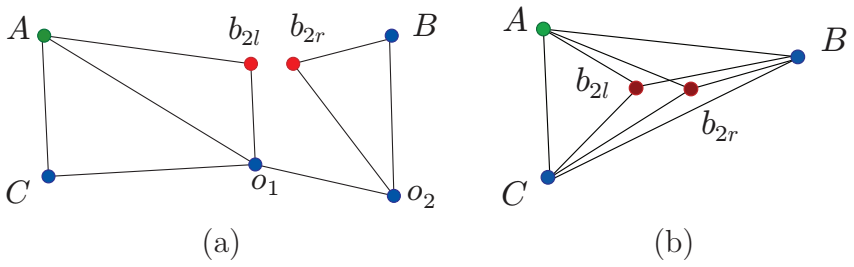


Figure 4.8: (a) One possible connectivity subgraph during (t_4, t_5) when mulitple agents are in the workspace. (b) Updated connectivity subgraph reflecting whether two vertices are reachable without triggering additional sensor recordings.

For the next recording, (b_2, t_4), three subgraphs need to be built, one starts from A, one starts from b_{1u}, and one starts from b_{1d}. Following the naming convention of G_1^0, these should have names G_4^0, G_4^{11}, and G_4^{12}, respectively. To build G_4^0, we need to keep vertices $\{A, b_{2l}, b_{2r}, o_1, o_2\}$. We also add $\{B, C\}$ since these are vertices of $\mathcal{C}_p$ that are reachable from

Algorithm 7 GETSUBGRAPHMULTI

Input: $G = (V, E)$, the start vertex v_s, $\mathcal{C}_p$, the active occupancy sensors O, and goal vertices V_G.

Output: $G' = (V', E')$, the part of G that is reachable from v_s.

```
1: V_C ← C_p ∪ O ∪ {s}
2: G' ← GETREACHABLESUBGRAPH(G, v_s, V_C, V_G)
3: for all o ∈ (O ∩ V') do
4:     add to E' an edge between each pair of o's neighbors
5:     remove o from G'
6: end for
7: return G'
```

$\{A, b_{2l}, b_{2r}, o_1, o_2\}$ without crossing additional sensors. This gives us the subgraph in Fig. 4.8(a). To facilitate searching, for each pair of neighbors of an active occupancy sensor, we add an edge between them and remove the occupancy sensor vertex, which yields the graph in Fig. 4.8(b). The pesudocode for building this subgraph, GETSUBGRAPHMULTI, is given in Algorithm 7. Assuming we have obtained G_4^{11} and G_4^{12} similarly, the next sensor recordings is (o_2, t_5), which corresponds to the deactivation of o_2. For this event, we need to create five subgraphs starting from $A, b_{1u}, b_{1d}, b_{2l}, b_{2r}$ with names $G_5^0, G_5^{11}, G_5^{12}, G_5^{41}, G_5^{42}$, respectively. Since during $[t_5, t_f]$, no new locations of G become reachable and no other beam sensor recordings happen, this part of $\mathbf{s}$ can be ignored. After connecting all these subgraphs based on sensor crossings, we obtain the composite graph G_s as illustrated in Fig. 4.9.

Before continuing with searching G_s, we make one observation from the aforementioned graph building procedure: Since each sensor recording may cause up to $O(m)$ new subgraphs to be built, up to $O(m^2)$ subgraphs may be built altogether. This is the number of subgraphs in G_s since each subgraph appears once in G_s. To search through G_s for a path matching $\mathbf{p}$, the same strategy from VALIDATEAGENTSTORY can be applied. That is, a dynamic programming approach can be used in which a subproblem is a tail of $\mathbf{p}$ and a location in G_s. Since there are no more than $O(m^2)$ copies of p_i in G_s, there can be at most $O(m^2)$ subproblems after each p_i is processed. Similar to VALIDATEAGENTSTORY, during the

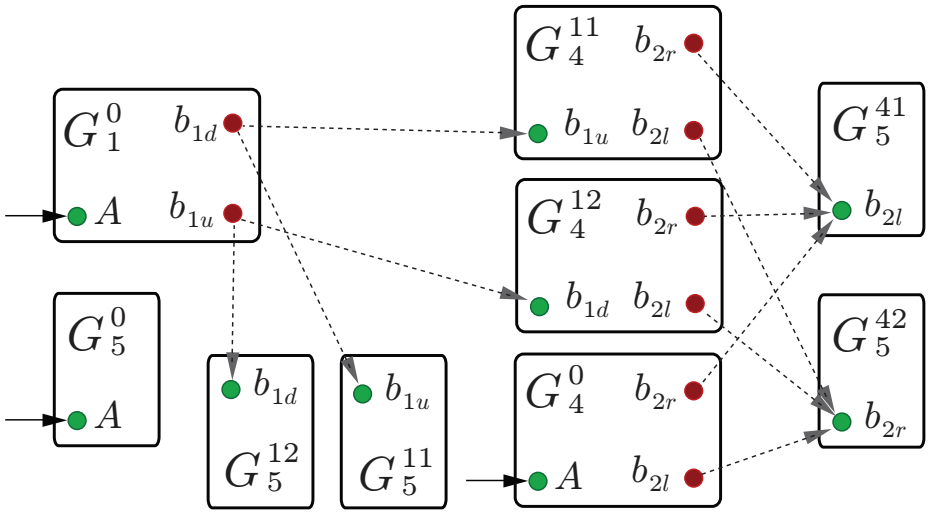


Figure 4.9: Sketch of the composite graph G_s.

processing of each p_i each subgraph only needs to be considered once. This limit the time complexity of searching G_s at $O(nm^2 \log m_w)$. To get the total time complexity, we need the time for building G_s, which is m^2 times the cost of the subroutine GetSubGraphMulti. The running time of GetSubGraphMulti is determined by the loop at lines 3 through 6, which takes $O(m_w^3)$ time. This yields the overall time complexity $O(m^2(n \log m_w + m_w^3))$. Since the algorithm operates much like ValidateAgentStory, we omit the pseudocode.

4.4 The Approximate Path Inference Problem and Its Solution

Up until now, we introduced and solved problems in which an agent's account of its path in an (single/multiple agent) environment is validated against recordings from a sparse network of sensors deployed in the same environment. In that work, an agent's story is required to be error-free and has start/end time matching those of the sensor recordings. Such formulations are restrictive for two reasons: (1) even a truthful agent is likely to introduce errors in recalling a long story, especially when the agent in question is a human, and (2) requiring matching start/end time can be hard to guarantee. Moreover, when an agent's story is not consistent with an observation history, the algorithms we have introduced so

far do not provide an alternative path for the agent that is "close" to the agent's story. In this section, we generalize the Problem 1 to define and address *approximate path inference* problems to remove all these limitations. We focus on the single-agent workspace and later discuss additional formulations involving multiple agents.

4.4.1 Approximate path inference problems

The formulation of Problem 1 is quite restrictive in at least two ways. First, requiring the story and the observation history to span the exact same time interval $[t_0, t_f]$ may not be always possible. For example, the sensor network may be inactive at some point due to daily operation schedule or system maintenance. We capture and generalize this case with the following problem.

Problem 3 (Exact story with mismatching time intervals) *Let there be a single agent in a workspace. The agent provides a story* $\mathbf{p}$ *for the time interval* $[t_0, t_f]$*. The sensors in the workspace produce an observation history,* $\mathbf{s}$*, for a time interval* $[t'_0, t'_f]$ *that overlaps with* $[t_0, t_f]$*. Validate whether* $\mathbf{p}$ *is consistent with* $\mathbf{s}$*. Extract a feasible path if the story is consistent.*

The second restriction is that a truthful agent, be it human or robot, may inevitably make mistakes in recalling a story. One very likely scenario here is that the agent may have missed in $\mathbf{p}$ locations in $\mathcal{C}_p$ it has visited. For instance, a truthful agent may have recalled a story $\{A, B, C\}$ although its story is actually $\{A, B, A, C\}$. This scenario translates into the following formulation (for two strings/sequences $\mathbf{p}, \mathbf{p}'$, the expression $\mathbf{p} \leq \mathbf{p}'$ or $\mathbf{p}' \geq \mathbf{p}$ denotes that $\mathbf{p}$ is a subsequence of $\mathbf{p}'$).

Problem 4 (Partial story with matching time intervals) *Let there be a single agent in a workspace. The agent provides a story* $\mathbf{p}$ *for the time interval* $[t_0, t_f]$*. During the same time interval, the sensors in the workspace produce an observation history,* $\mathbf{s}$*. Let* $\mathbf{p}' \geq \mathbf{p}$ *be*

a sequence with elements from $\mathcal{C}_p$. Validate whether there exists a $\mathbf{p}'$ *that is consistent with* $\mathbf{s}$*. If such a story* $\mathbf{p}'$ *exists, find one of shortest length.*

Problem 4 motivates a more general case: The agent, even with best effort, may add locations it has not visited (insertion), miss locations it has visited (deletion), or report some locations it has visited incorrectly (substitution). As an example, a truthful agent may have recalled $\{A, B, C\}$ when the actual story is $\{C, B, B, D\}$. Calling each of the three possibilities to introduce an error as an *edit*, we obtain the following problem.

Problem 5 (Error in story with matching time intervals) *Let there be a single agent in a workspace. The agent provides a story* $\mathbf{p}$ *for the time interval* $[t_0, t_f]$*. During the same time interval, the sensors in the workspace produce an observation history,* $\mathbf{s}$*. Let* $\mathbf{p}'$ *be a sequence with elements from* $\mathcal{C}_p$*. Find a* $\mathbf{p}'$ *that is consistent with* $\mathbf{s}$ *such that* $\mathbf{p}$ *can be obtained from* $\mathbf{p}'$ *with the least number of edits.*

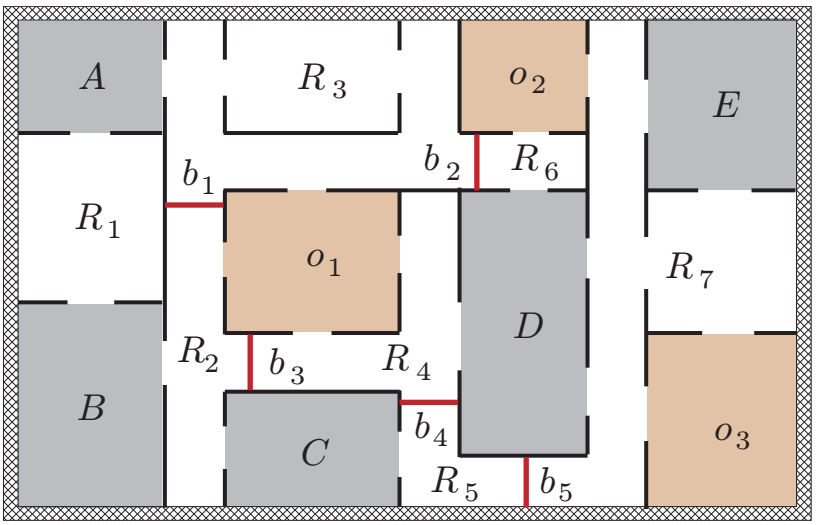


Figure 4.10: A workspace with five beam detectors b_1 through b_5, three occupancy sensors o_1 through o_3, and five labeled rooms A through E. Thus, $\mathcal{C}_p = \{A, B, C, D, E\}$, $\mathcal{C}_s = \{b_1, b_2, b_3, b_4, b_5, o_1, o_2, o_3\}$. There are seven connected components R_1 through R_7 when regions guarded by sensors and rooms are treated as workspace obstacles.

We point out that besides the above problems, many other interesting formulations are possible. For example, one may require the suspect's story have a maximum error rate of

10%, for which a solution can be derived from the solutions of Problem 5. Due to limited space, we focus on the general cases given in Problem 3 through 5; additional extensions are then discussed later. In presenting the solutions to Problem 3 through 5, we update the example to that given in Fig. 4.10 since the example given in Fig. 1.6 does not possess enough structure for the presentation. The connectivity graph of the environment from Fig. 4.10 is given in Fig. 4.11.

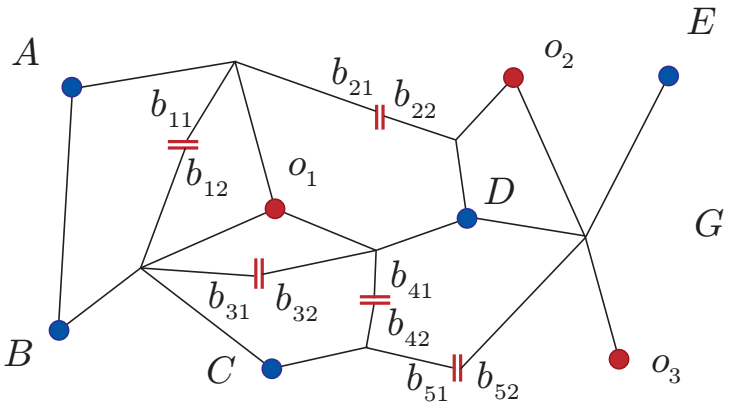


Figure 4.11: The connectivity graph G for the example given in Fig. 4.10.

4.4.2 Story and observation history with mismatching time intervals

For Problem 3, it turns out that we can simply use ValidateAgentStory as a subroutine to solve it. the problem has several subcases, depending on how $[t_0, t_f]$ and $[t'_0, t'_f]$ compare. Assuming that t_0, t_f, t'_0, t'_f are pairwise different, the following six cases are possible:

$$t_0 < t_f < t'_0 < t'_f, \quad t_0 < t'_0 < t_f < t'_f,$$
$$t'_0 < t_0 < t_f < t'_f, \quad t_0 < t'_0 < t'_f < t_f,$$
$$t'_0 < t_0 < t'_f < t_f, \quad t'_0 < t'_f < t_0 < t_f.$$

For the first and last cases, $[t_0, t_f]$ and $[t'_0, t'_f]$ are disjoint. In these cases, the sensors cannot possibly observe the agent during $[t_0, t_f]$; nothing needs to be done about $\mathbf{p}$ (as long as $\mathbf{p}$

does not conflict with itself). We are not yet done, because we still need to check whether an empty story is consistent with $\mathbf{s}$. This can be done using ValidateAgentStory (The search portion only takes $O(m \lg n_w)$ time).

The second case is $t_0 < t'_0 < t_f < t'_f$, which means that a later portion of the agent's story overlaps with an earlier portion of the sensor observation history. This can be handled by running ValidateAgentStory algorithm n times. For the i-th run, the story is $(p_i, \ldots, p_n)$ and the sensor observation history is $\mathbf{s}$. If any of the runs returns true, then the story is valid; otherwise the story is inconsistent. We note that the search can be arranged more efficiently by working on the same p_i's of different runs at the same time; thus, the time complexity for this case remains $O(nm \lg n_w)$.

For the third case, $t'_0 < t_0 < t_f < t'_f$, the story $\mathbf{p}$ may start in the middle of G_s. ValidateAgentStory can be easily modified to handle this: Instead of starting in G_1, we now allow the search to start at (p_1, j) for all applicable j's. If p_n is ever reached in the search, the modified algorithm should report that $\mathbf{p}$ is consistent with $\mathbf{s}$. The time complexity again remains the same. Following along the same lines, we can decide cases four and five.

4.4.3 The composite automaton structure

Although solution to Problem 3 is a relatively straightforward extension of the algorithm ValidateAgentStory, it is not immediately clear whether the approach applies to Problems 4, 5, and more general cases. Part of the difficulty comes from the composite graph G_s: It has more structure than a standard directed graph with $O(mn_w)$ vertices. On the other hand, we observe that ValidateAgentStory operates much like an automaton in the sense that it processes $\mathbf{p}$ sequentially and either accepts or rejects. This prompts us to ask: Can we turn G_s into an automaton and apply results from automata theory to tackle our problem? We answer the first part of this question in this section and delay the second part to the next.

The conversion of a connectivity graph G and a observation history $\mathbf{s}$ into finite automata is relatively straightforward. For each pair of consecutive sensor recordings s_i, s_{i+1} (except when $i = 0, m$), we isolate the part of G that can reach vertices of s_{i+1} from vertices of s_i and convert it to an automaton. When $i = 0$, we let the start state of the automaton transit to all vertices in $\mathcal{C}_p$ and when $i = m$, we let all vertices in $\mathcal{C}_p$ be acceptance states. These automata are then chained together to give a composite automaton.

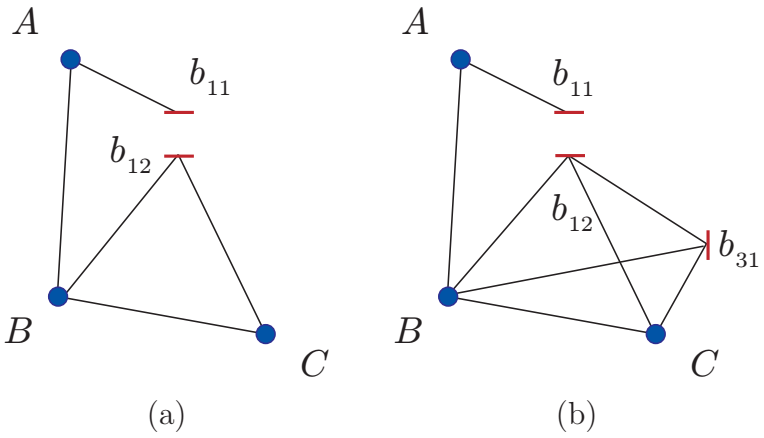


Figure 4.12: Available subgraphs induced by sensor recordings. (a) The available subgraph of the example from Fig. 4.10 when b_1 is the first sensor recording. (b) The available subgraph of the example from Fig. 4.10 for the recording $b_1 b_3$.

As a concrete example, we assume again that the observation history is $\mathbf{s} = b_1 b_3 o_2 o_2 b_4$. The first sensor recording is b_1, which means that agent x must at one point reach either b_{11} or b_{12} and cross the beam detector. From Fig. 4.11 it is clear that if agent x starts from D or E, it is impossible for the agent to get to b_1 without triggering other sensors. Therefore, we only need to consider A, B, C, b_{11}, and b_{12}. Coincidentally, this gives us that part of G available is the same as Fig. 4.12(a). We may convert this to the automaton in Fig. 4.13(a).

The sensor that is activated next is b_3. Since agent x must start from b_{11} or b_{12}, the connectivity graph G tells us that the agent can only pass b_3 from left to right. The part of G available in this case is the same as 4.12(b). Follow similar construction, we obtain the corresponding automata shown in Fig. 4.13(b). Note that Fig. 4.13(b) represents

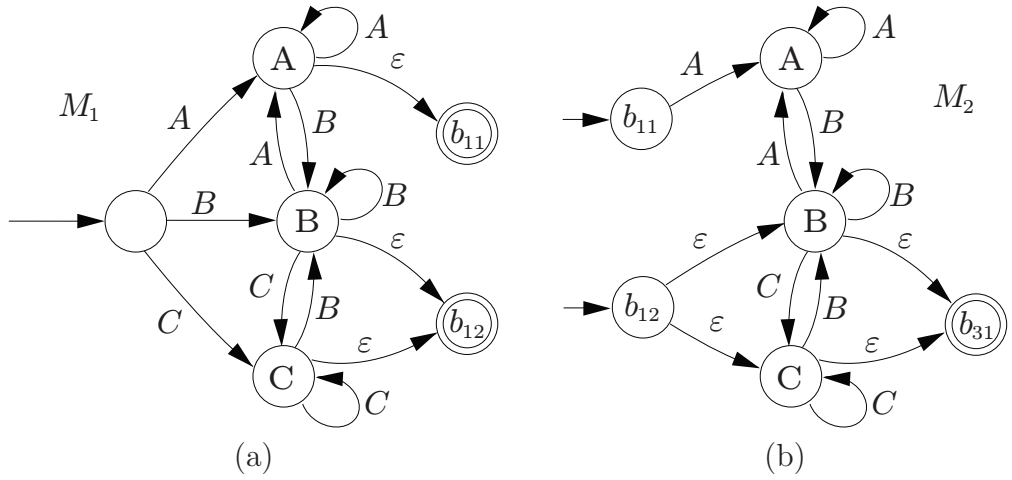


Figure 4.13: The corresponding nondeterministic finite automaton. (a) The corresponding NFA when b_1 is the first sensor crossed. (b) The NFA for consecutive sensor recordings $b_1 b_3$.

two automata with one starts at from b_{11} and one starts from b_{12}. The rest of the sensor recordings from **s** can be processed in the same manner. For an observation history **s** of length m, $m+1$ automata are obtained. Denote these automata M_1 through M_{m+1}. For implementation purpuse, we connect all states of M_{m+1} to a single acceptance state F via an ϵ transition. It is straightforward to observe that a story **p** is consistent with **s** if and only if the story string can be partitioned into pieces $P_1, \ldots, P_{m+1}$ such that P_i is accepted by M_j. Alternatively, if we connect M_1 through M_{m+1} together to get a *composite automaton*, M (Fig. 4.14), then **p** must drive M from start state to F.

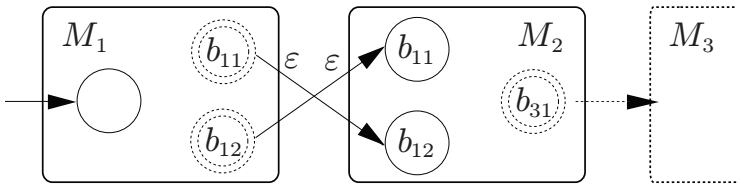


Figure 4.14: Sketch of the composite automaton for the observation history string $b_1 b_3 o_2 o_2 b_4$.

With minimal effort, ValidateAgentStory can be modified to work with the composite automaton M, which will be able to resolve Problems 1 and 3, keeping the dynamic programming framework intact. Although it does not make the algorithm more efficient, the

approach provides an alternative interpretation of these problems: Searching $\mathbf{p}$ through G_s now becomes simulating M over $\mathbf{p}$, making these problems tests of whether a string belongs to a regular language.

4.4.4 Partial story

For Problems 1 and 3, only a few stories are checked against an observation history $\mathbf{s}$. This is no longer the case for Problems 4 and 5 since an infinite number of possible stories must be checked in the process of solving these problems. This is where the automaton structure comes in: If we simulate a story $\mathbf{p}$ over an automaton, we know that on seeing p_i, there are a set of fixed states the automaton can be in. As long as we maintain these finite number of states, an infinite number of string patterns can be handled. In fact, existing results from automata theory have completed part of the job for us: Problems 4 and 5 can be viewed as the *string edit distance problem between a string and an automaton* (with the string being $\mathbf{p}$ and the automaton being M), for which efficient algorithms exist.

Our job is not done, however. Exisitng algorithms assume that the string $\mathbf{p}$ and the automaton M are the inputs. In our problem, the automaton M is an intermediate input built from $\mathbf{s}$ and G_s; thus, a direct adoption of approximate string matching algorithms may not fully explore the structure of M. In this section, we explore how the sequential nature of M allows us to subdivide Problem 4 more effectively than off-the-shelf approaches. After solving Problem 4, we sketch at high level how the same structure helps solve Problem 5 as well.

We continue to work with the example from Fig. 4.10 and assume that the story is $\mathbf{p} = ABDEC$ and the observation history is $\mathbf{s} = b_1 b_2 o_2 o_2 b_4$. Suppose that we obtain the automata M_j's as well as M from $\mathbf{s}$ already. With the analysis from Section 4.4.3, Problem 4 becomes finding a shortest $\mathbf{p}' \geq \mathbf{p}$ such that $\mathbf{p}'$ is accepted by M. The problem may seem a bit daunting at first glance: An infinite number of $\mathbf{p}'$ needs to be examined, as long as

$\mathbf{p}' \geq \mathbf{p}$.

However, any consistent $\mathbf{p}'$ must drive the composite automaton M to an accepting state. If we assume that some $\mathbf{p}' \geq \mathbf{p}$ is among the shortest strings accepted by M, then it must have the format $\mathbf{p}' = \omega_1 A \omega_2 B \omega_3 D \omega_4 E \omega_5 C \omega_6$, in which ω_i's denote the parts missing from $\mathbf{p}$. Since $p_1 = A$, we search M and find shortest strings from the start state of M to all copies of A in M, there are up to $m + 1$ of these. We denote these strings as $\sigma(1, 0, j)$ and the copy of A from M_j as (A, j). For each j there may be multiple such strings of the same shortest length; in this case any of these will do. By the principle of dynamic programming $|\omega_1| + 1 = |\sigma(1, 0, j)|$ for some j. Moving to $p_2 = B$, for each (B, k) let us denote a shortest string taking M from some (A, j) to (B, k) as $\sigma(2, j, k)$. Among these $\sigma(2, j, k)$'s for a fixed k, we need to get one such that $\sigma(1, 0, j) + \sigma(2, j, k)$ is a shortest. Again, the principle of dynamic programming tells us that for some j, k, $|\omega_1| + 1 = |\sigma(1, 0, j)$ and $|\omega_2| + 1 = \sigma(2, j, k)|$. Following this method, if some simulation branches remain after all of $\mathbf{p}$ are exhausted, then a consistent $\mathbf{p}'$ exists and one can be extracted via backtracking the search process.

Given the analysis, it becomes clear that the insight enabling the reduction of a factor of m in ValidateAgentStory also applies here. That is, in finding the shortest string containing $p_1 \dots p_i$ and taking M from the start state to (p_i, k), we do not need to look at (p_{i-1}, j) for all $j \leq k$. Instead, we only need to know the shortest string that takes $(p_{i-1}, j), j \leq k - 1$ for some j, to the start state(s) of M_k; the rest of the search is limited to M_k. Since searching inside M_k for the shortest path can be done with Dijkstra's algorithm in $O(n_w^2)$, the overall running time for the search part of validating a partial story is $O(nmn_w^2)$. The pseudocode is described in Algorithm 8. Note that we append to $\mathbf{p}$ an element F, which is the acceptance state of M. Element $L(i, j)$ of the 2D array L stores the length of the shortest string that drives M to the state (p_i, j) and contains $p_1 \dots p_i$ as a subsequence. The subroutine ShortestLen(a, b) returns the length of the shortest string that takes M from

Algorithm 8 ValidatePartialStory

Input: $\mathbf{p} = (p_1, \ldots, p_n, F), M, M_1, \ldots, M_{m+1}$
Output: **true** if M accepts some $\mathbf{p}' \geq \mathbf{p}$, **false** otherwise

```
1: initialize 2D array L as array of ∞'s
2: L(0,1) ← 0
3: for i = 1 to n + 1 do
4:    for j = 1 to m + 1 do
5:       ℓ ← ∞
6:       for each start state S_k of M_j do
7:          t ← {min_{j'} ShortestLen((p_{i-1}, j'), S_k)}
8:          ℓ ← min{ℓ, t + ShortestLen(S_k, (p_i, j))}
9:       end for
10:      L(i,j) ← min{ℓ, ShortestLen((p_{i-1}, j), (p_i, j))}
11:   end for
12: end for
13: if L(n + 1, m + 1) ≠ ∞ then
14:    return true
15: end if
16: return false
```

state a to state b. As discussed, we can obtain ShortestLen$((p_{i-1}, j'), S_k)$ incrementally.

4.4.5 Story with errors

Although Problem 5 appears harder than Problem 4, it is not clear that it is more time consuming. On one hand, to allow insertion, deletion, and substitution of story string $\mathbf{p}$, we need to know the shortest edit distance to reach all states of M for each p_i, instead of knowing only the shortest distance to states (p_i, j). Denote this distance $D(p_i, T)$ in which T is a state of M and let $D(p_i, S, T)$ be the shortest edit distance from state S (of M) to T on p_i, we obtain the recursion

$$D(p_i, T) = \min_S \{D(p_{i-1}, S) + D(p_i, S, T)\}.$$

For a general automaton of Q states, $D(p_i, S, T)$ requires $O(Q^3)$ ($O(m^3 m_w^3)$ for our problem) computation time using a modified all pairs shortest path algorithm [97]. In our case, since

$D(p_i, S, T)$ is ∞ for $S \in M_k, T \in M_j$ when $j < k$, we may subdivide the calculation of $D(p_i, T)$ further, staged at each M_j. Suppose T is an internal state of M_j (not start/end states), $\{S_j\}$ are the start states of M_j (there are at most two of these), then the shortest edit distance from $S \in M_{j-1}$ to T, passing some $\{S_j\}$ can be obtained as

$$\min_S\{D(p_{i-1}, S) + \min\{\min_{S_j}\{D(p_i, S, S_j) + D(\epsilon, S_j, T)\}, \min_{S_j}\{D(\epsilon, S, S_j) + D(p_i, S_j, T)\}\}\}.$$

We can regroup the formula as

$$\min_{S_j}\{\min\{\min_S\{D(p_{i-1}, S) + D(p_i, S, S_j)\} + D(\epsilon, S_j, T),$$
$$\min_S\{D(p_{i-1}, S) + D(\epsilon, S, S_j)\} + D(p_i, S_j, T)\}\}.$$

Hence, instead of $O(m^3 m_w^3)$ time, the sequential structure of M enables us to cut the processing time per p_i to at most $O(m m_w^3)$.

On the other hand, unlike in Problem 4, with the introduction of deletion and substitution, a matching story is always possible for Problem 5, as long as the sensor recordings are self-consistent. That is, the language of M is not empty. In the worst case, we can change $\mathbf{p}$ to the shortest string accepted by M via substitution, followed by insertion if $\mathbf{p}$ is too short and followed by deletion if $\mathbf{p}$ is too long. If we denote the length of the shortest string accepted by M as n', then a $\mathbf{p}'$, accepted by M and closest to $\mathbf{p}$, cannot have length more than $\max\{n, n'\}$. This is also the maximum number of edits necessary.

From the preceeding observation, we see that it is not necessary to carry all pairs of shortest path search over M for each p_i. This observation is the same one that enables the approach from [101] to keep only two levels of a transducer structure in memory. We adapt the same transducer construction to our problem, which has the high level structure illustrated in Fig. 4.15. Denote this transducer U. Comparing U with that from [101], our transducer has additional structures: It is directed not only from top to bottom but also

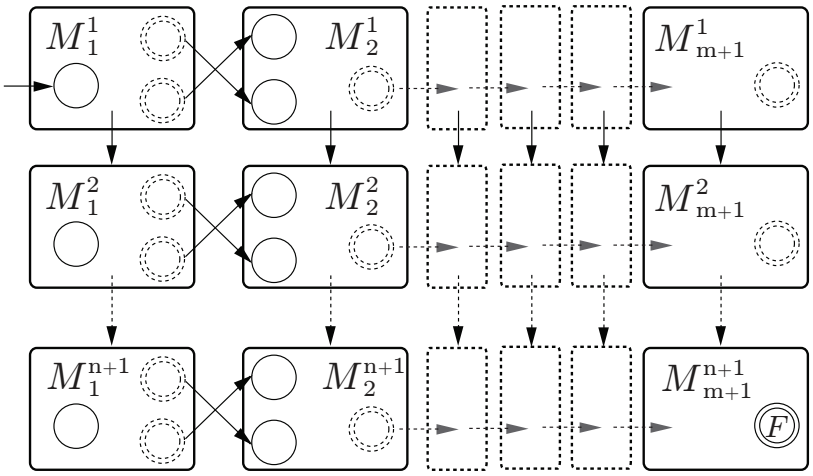


Figure 4.15: Sketch of the transducer built from string $\mathbf{p}$ and automaton M.

from left to right. This means that searching U can be partitioned into searching individual M_j^i's from left to right then top to bottom. For Problem 4, finding $\mathbf{p}'$ with the smallest number of edits is equivalent to finding the accepting string of U with the smallest cost. This is then a single source shortest path problem on U. As said, for each i, we carry out the search from left to right from M_1^i to M_{m+1}^i, which takes time $O(m \cdot m_w \cdot m_w \lg m_w)$. Doing this for all i (top to bottom) then takes time $O(nmm_w^2 \lg m_w)$. In contrast, preprocessing alone takes $O(m^3 n_w^4)$ in [97] (recall that m, n are of comparable lengths). Our result is also slightly better than the (more general) algorithm presented in [101], which has time complexity $O(nmm_w^2 \lg(mm_w))$ in this context.

4.4.6 Additional extensions

Having provided algorithms for Problems 3 through 5, many additional questions can be answered. We briefly discuss some of the possible extensions here.

Constraints. Besides the restrictions placed in Problems 1 through 5, many other problem variations are also possible and can be resolved similarly. The case that requires the suspect's story having a maximum error rate of 10% is one such constraint. As another example, the agent may be confident about certain parts of its story. In Problem 4, this

translates to some substrings of the story string must be processed without inserting additional locations. Yet another example is that the story and/or the observation history may contain time "gaps," as an extension of Problem 3. These additional constraints can be dealt with by modifying the aforementioned algorithms.

Multiple agents. What if there are multiple agents in the workspace? As indicated by earlier analysis, when an unknown number of agents are in the workspace, not all sensor recordings are triggered by agent x; there are $2^{|\mathbf{s}|}$ possible observation histories by agent x. Moreover, the observation history has a different structure: The active occupancy sensors act as hubs that connect parts of the workspace together. Any agent can then pass an active occupancy sensor multiple times. Carrying over the analysis, we observe that the "unit automaton structure" still holds: A composite automaton can be built with individual automata having no more than $O(m_w)$ states. Such orthogonality allows direct combination of the multiple agent case from earlier analysis with the solutions of Problems 3 through 5, generalizing them to include multiple agent cases.

CHAPTER 5

OPTIMAL SENSOR PLACEMENT FOR PATH INFERENCE AND PREDICTION

In Chapter 4 , we have assumed that the sensors' placement is given to us. This assumption is often valid in indoor environments where surveillance equipments (sensors) are already set up. However, if we have the opportunity to place the sensors, a natural *optimal sensor placement* problem comes up: How can we place the sensors to complete the path inference task at hand while minimizing the cost of the equipment? In this chapter, we investigate the problem of optimally allocating sensors to monitor agents as they move from one location to another, carrying out unknown tasks in these locations. That is, we seek sufficient and/or necessary placements of sensors that produce different outputs for different (homotopy classes of) paths. This problem can be viewed as the online version of the path inference problems discussed in Chapter 4. Our focus will be on tracking and predicting the behavior of a single agent assuming that the agent travels between point of interests following shortest paths. For this problem, we conjecture that the problem itself is NP-hard and provide an polynomial time algorithm for computing a solution with a cost that is at most twice of the optimal solution (2-approximation).

5.1 Path Inference Problems

When it comes to deploying a network of sensors to monitor agents' behaviors, an optimization problem naturally arises. Since stronger sensors usually cost more, it is desirable to use a minimal sufficient set of weaker sensors in place of stronger ones, if possible. This requirement is even more relevant in a sensor network setting: As more sensors are added to

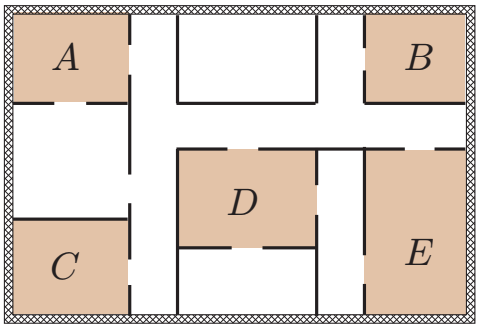


Figure 5.1: An indoor environment with five rooms of interest.

a network, more computation and communication resources are needed, which may overload the entire system. Even if there are enough communication and computational resources available, knowing the minimal set of sensors that are sufficient for the path inference task is still beneficial. Turning off the sensors that are not needed will help save resources (such as energy and communication bandwidth) and improve system response time.

To make sense of the problem formulation that will follow, let us consider an example in an environment similar to that used in Chapter 4, given in Fig. 5.1. Assume that we are interested in monitoring agent activities in rooms A - E. Assuming that the single agent that we are tracking is rational and does not perform useless work (i.e., it does not go out a room and then go back to the same room or take unnecessarily detours), the possible paths taken by the agent induces a connectivity graph structure over the environment. Depending on the types of sensors (e.g., occupancy sensors, beam detectors, and so on) and the environment's features, the graph structure may vary greatly. For example, for the graph given in Fig. 5.2(a), it is possible to install an occupancy sensor at a red vertex and a beam detector between each pair of black vertices.

Clearly, if we install occupancy sensors at each of the five rooms, then it is possible to distinguish all possible agent paths of interest. Note that we can replace an occupancy sensor in a room with beam detectors at the doorways of that room. Alternatively, the same can be achieved by installing beam detectors as given in Fig. 5.2(b), as well as occupancy sensors

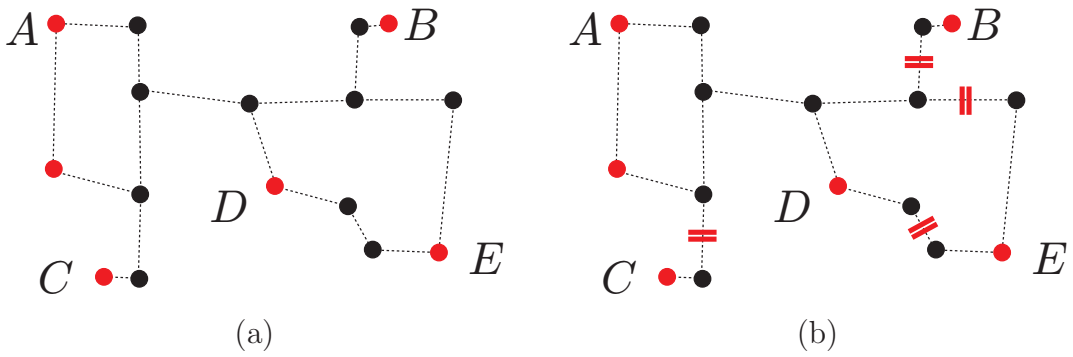


Figure 5.2: (a) One possible graph structure capturing the connectivity of the environment. (b) A possible sensor setup.

in rooms A and D. This latter solution, in additional to have potential cost advantage over the former one, also allows *prediction* of the agent's future behavior. For example, if the agent left room A and then the beam detector outside of room C is triggered, we can predict that the agent is going to room C. If the building is designed carefully with sensors placed correctly, it is then possible to predict the behavior of a rational agent fairly easily.

Similar problems also arise in outdoor environments; one such example is the placement of weighing stations on highways to monitor freight trucks' load as they transport goods from city to city. Given that these trucks are mostly used to transport goods between large cities, we may reasonably assume that a truck's load is roughly constant as it goes from one city to another. If we want to weigh the truck for every inter-city trip it makes, we need to set up weighing stations along the highways. If we imagine that the rooms in Fig. 5.2 are the cities of interest and other nodes are small towns and hubs, we again obtain a sensor placement problem. In this case, however, we can only setup sensors along the edges (similar to beam detectors). Since it is in the truck companies' best interest to save on transportation expenses, we may further assume that a truck generally follows a shortest path between any two cities.

These practical scenarios inspire us to study the problem of optimal sensor placement for path inference and prediction assuming that the agent always follows shortest paths. We

now formally describe this as an optimization problem on a graph.

5.1.1 A graph based optimal sensor placement (OSP) formulation

Let $G = (V, E)$ denote a connected, undirected graph with $V = \{v_i\}$ and $E = \{e_i = (u_i, v_i) : u_i, v_i \in V\}$ being the vertex set and edge set of G, respectively. Let $V_r \subset V$ be a subset of the vertices of G. Let there be one or more agents traveling on G. At any given time, an agent tries to move from some vertex $u \in V_r$ to some vertex $v \in V_r$, following an optimal path between u, v. To measure the cost for an agent to pass an edge, we assign each edge a value via the map

$$d : E \to \mathbb{R}^+, \quad e_i \mapsto d_i. \tag{5.1}$$

For example, d_i may represent the physical length of edge e_i, in which case an optimal path for an agent is a shortest path. For two vertices u, v, we also use $d(u, v)$ to denote the same cost of crossing the edge.

Assume that we want to monitor the behaviors of the agents via a set of sensors, $\mathbf{s} = \{s_i\}$, which can be installed on any edge of G. For example, these sensors may be security cameras which are capable of identifying a target's identity and moving direction as an agent passes through the regions monitored by these cameras. To account for the cost of sensor installation/operation, we associate with each edge of G a numerical value via the map

$$c : E \to \mathbb{R}^+, \quad e_i \mapsto c_i. \tag{5.2}$$

In network algorithms, the cost as defined in 5.2 is frequently referred to as *capacity*, which we sometimes use to mean the same thing when more appropriate. Given the setup so far, the basic question we ask in this chapter is the following.

Problem 1 (Optimal sensor placement (OSP)) *Given a 4-tuple (G, d, c, V_r), choosing a set of edges $C \subset E$ to place sensors such that*

1. *For each $v \in V_r$ that an agent decides to visit next, at least one of the sensors can detect this event before the agent reaches v.*

2. *The total cost of sensors (i.e., capacities on edges) is minimized.*

5.1.2 An example OSP problem and its solution

To offer a concrete example of the OSP problem and its solution, let us look at the problem given in Fig. 5.3. The vertex set we are interested in guarding is $V_r = \{u, v, w, x\}$. The numbers on the edges of the graph denote the length of the edges; assume that the capacities are the same on all edges. For this graph, the edge set $\{vw, vx, wu, ux, du, dx\}$ is an optimal solution. We must have vw in any solution since this is the shortest path between v and w. Same is true for vx, wu and ux. We must have du in the cut set because the minimal candidate edge sets required to guard the shortest paths between v, u are $\{va, vb\}, \{va, bc\}, \{ac, vb\}, \{ac, bc\}$, and $\{du\}$. We cannot use cd since it is also on the shortest path between w and x. Finally, we may add either dx or wc to guard the path $wcdx$.

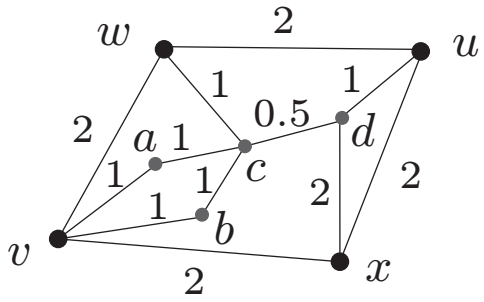


Figure 5.3: An example graph with $V_r = \{u, v, w, x\}$. All edge lengths are marked on the figure; all edges have the same capacity.

5.1.3 Relation to the minimum cut problem

Since a solution to the OSP problem is a set of edges of a graph G, a very natural idea toward obtaining a solution is to relate the problem at hand to the minimum edge cut problem. We briefly state a few related problems here without formally describe these problems; more details will follow in later sections. Given a graph $G = (V, E)$ with non-negative edge capacities, the well-known *max-flow min-cut* theorem states that a *minimum edge cut* (minimum s, t-cut) of the graph between two of its vertices (say, s, t) has a total edge capacity equaling the *maximum flow* (max-flow) from s to t through the same graph, not exceeding the given edge capacities. Since efficient algorithms exist for the max-flow problem [108], an efficient algorithm also exists for the minimum s, t-cut problem (this is a classical result in integer linear programming). The algorithm also works on directed graphs (directed minimum s, t-cut). However, the edge cut problem becomes NP-hard for a pair of vertices on a directed graph when the cut is for both directions (the directed minimum multiway cut) [109] or three (or more) vertices on an undirected graph (the multiway cut problem) [110].

Given that no polynomial time algorithm exists for the directed minimum multiway problems unless P = NP, the only polynomial result on minimum cut is for the (directed and undirected) minimum s, t-cut problem. If we restrict our attention to a pair of vertices in V_r, although our formulation mimics that of the minimum s, t-cut, there are at least two significant differences:

1. We are only interested in edges that appear on some shortest paths between two vertices of V_r, which are not all edges of G.

2. Our problem demands a cut among many vertices of a graph but the min-cut problem only deals with cuts between two vertices.

These differences suggest that we cannot directly adapt the solution to the minimum s, t-

cut problem to solve our problem even when $|V_r| = 2$. Nevertheless, the minimum s, t-cut algorithm proves useful for obtaining the 2-approximation solution. Before getting to the solution, we first study the structure over the E induced by the assumption that an agent must follow shortest paths between vertices of V_r.

5.2 Edge Pruning and Partitioning

5.2.1 Pruning irrelevant edges

Given an instance (G, d, c, V_r) of the OSP problem, we observe that some edges of the graph may not be on any shortest path between two vertices of V_r. These edges can be safely ignored. To get rid of these *irrelevant* edges, we may run Dijkstra's algorithm for each vertex $v \in V_r$. If an edge does not appear on any shortest path between two vertices $u, v \in V_r$, then that edge is simply removed from E. Assume that for the rest of the chapter we work with a connected G such that each edge of G is on a shortest u, v-path for some $u, v \in V_r$.

5.2.2 Sensor induced edge partition

For each edge $e \in E$ that remains after pruning, it may be on one or more shortest u, v-paths for some $u, v \in V_r$. This information can be obtained during the pruning process. If e is on a shortest path between two vertices u, v with $u, v \in V_r$ such that no other vertex $w \in V_r, w \neq u, v$, is also on the same u, v path, then we associate e with u and v. Based on the shortest paths an edge lies on, we may put it into one of the following three mutually exclusive partitions:

1. The edge is associated with exactly two vertices $u, v \in V_r$. That is, the edge lies on some shortest u, v paths for exactly one pair of $u, v \in V_r$. We put these edges in the edge set E_{uv}.

2. The edge lies on multiple shortest paths such that one terminals of these paths are the same (say the terminal is u) but the other terminals are not all the same (the edge must be associated withe more than two vertices of V_r). We put these edges in the edge set E_u.

3. The edge lies on a u, v shortest path and on a w, x shortest path such that the vertices u, v, w, x are four distinct vertices of V_r.

The mutual exclusivity is obvious from the definition. As an example, let us look at Fig. 5.3. In this graph, edge vw is of the first type, belonging only to shortest path between v, w. The same is true for edges wu, ux, vx, du, dx. Edge vc is of the second type, belonging to shortest paths vcw and $vxdu$. The same is true for wc. Edge cd is of the last type, belonging to shortest paths $wcdx$ and $vcdu$.

5.2.3 Properties of the partitioned edges

Let us look at a single vertex $u \in V_r$ and assume that the agent starts traveling from u to some vertex $v \in V_r, v \neq u$. For each edge the agent travels through during the process, the agent can only pass through the edge in a certain direction. This direction is called an *orientation* of the edge, which is more formally presented in the following proposition.

Proposition 2 *Given a connected undirected graph G and vertices $u, v_1, v_2 \in V_r$, if we orient all edges of all shortest paths between u and $v_i, i = 1, 2$ from u to v_i, then no edge on these paths may have two orientations.*

Proof. Assume that an edge ab has both orientations. Consider the case in which ab has different orientations on a shortest u, v_1 path and a shortest u, v_2 path. We may denote these paths as $P_1 := us_1abs_2v_1$ and $P_2 := us_3bas_4v_2$, respectively, in which $s_1 - s_4$ are continuous segments of paths (see e.g., Fig. 5.4). For simplicity, we assume that the two paths share a since edge; the same proof applies for multiple edges as well. Using len to denote the length

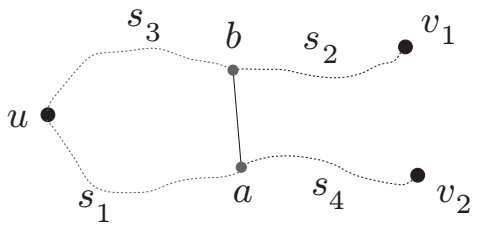


Figure 5.4: An edge ab belonging to multiple E_{uv_i} sets.

of an edge or a path, we have (by the fact that P_1, P_2 are shortest paths)

$$\text{len}(s_1) + \text{len}(s_4) \geq \text{len}(s_3) + \text{len}(ab) + \text{len}(s_4),$$
$$\text{len}(s_3) + \text{len}(s_2) \geq \text{len}(s_1) + \text{len}(ab) + \text{len}(s_2).$$

Adding up the two inequalities leads to an immediate contradiction. For the case where ab assumes different orientation on two different shortest u, v_1 paths, we may simply let $v_1 = v_2$ and the same proof applies. The same is true if ab assumes assumes different orientation on two different shortest u, v_2 paths. ■

Letting $v = v_1 = v_2$ yields the following corollary.

Corollary 3 *Given a connected undirected graph G and two vertices $u, v \in V_r$, if we orient all edges of all shortest u, v paths from u to v, then no edge may have two orientations.*

By Proposition 2 and Corollary 3, if an agent passes through an edge in some E_{uv} set, the agent must be heading to u or v. By Corollary 3, we can immediately tell whether the agent is going to u or v from its heading. Similarly, if an agent passes through an E_u edge in the direction of u, it must be moving to u. That is, the first two types of edges are potential locations for sensor installation. On the other hand, the last type of edge cannot be used for sensor installation since observing the passing of an agent along these edges cannot tell us where the agent is heading to.

5.3 A 2-Approximation Algorithm for the OSP Problem

5.3.1 A conjecture on the complexity of OSP

Given a directed graph $G = (V, E)$ with capacities on the edges and a set of k terminals, $s_1, \ldots, s_k$, a *multiway cut* is a set of edges whose removal disconnects every pair of terminals. That is, after the removal of the cut edges, there is no longer a directed path between any $s_i, s_j, 1 \leq i, j \leq k, i \neq j$. It was shown that the directed multiway cut is NP-hard for all $k \geq 2$ (Theorem 3.1 in [109]).

Theorem 4 *The problem of computing the minimum edge (node) multiway cut in directed graphs is NP-hard and max SNP-hard for all $k \geq 2$, even if all edge weights are 1.*

Due to the obvious similarity between the OSP problem and the multiway cut problem, our hypothesis on the complexity of the OSP problem is that the problem is NP-hard.

Conjecture 5 *The OSP problem is NP-hard.*

5.3.2 Computing a sufficient edge cut set

If the OSP problem is NP-hard, then the best we can hope for is a good polynomial time approximation algorithm unless P = NP. We now introduce, at a higher level, an efficient algorithm that yields a solution to the OSP problem that is a 2-approximation. That is, the edge set produced by this algorithm has an edge on each possible shortest path between any pair of vertices $u, v \in V_r$. The algorithm make repeated use of algorithmic solutions to the directed minimum s, t-cut problem, defined as follows.

Problem 6 (Directed Minimum s, t-cut) *Let $G = (V, E)$ be a connected, directed graph and $s, t \in V$. Let $c : E \to \mathbb{R}^+$ be the set of edge capacities. Find a set $C \subset E$ of edges that disconnects all directed s, t-paths such that $\sum_{e \in C} c(e)$ is minimal.*

The are many algorithms for solving the directed minimum s,t-cut problem and these algorithms are also asymptotically faster than the algorithm for the undirected minimum s,t-cut problem. Our algorithm essentially applies a directed s,t-cut algorithm for each vertex $u \in V_r$. To begin, we pick an arbitrary $u \in V_r$. To predict that an agent is moving to u, we need to set up sensors such that every shortest path with u as an endpoint is covered. From earlier analysis, the effective edges to setup sensors is the set

$$E'_u := E_u \cup (\bigcup_{u' \in V_r} E_{uu'}). \tag{5.3}$$

To get a minimum capacity edge set from E'_u for the task of predicting an agent is going to u, we perform the following operations:

1. Orient all shortest path with one endpoint u starting from u. By Proposition 2, no edge will have two different orientations. We then remove any edge of G that has no orientation. Note that all remaining edges are not in E'_u.

2. For edges that are not in the set E'_u, we change their capacities (costs for installing sensors) to infinity (a large number in practice), since we do not want them to be selected as cut edges.

3. Create a new terminal vertex t and for each vertex $u' \in V_r, u' \neq u$, add a (u', t) edge and orient it from u' to t. Assign infinite capacities to these edges.

4. For this oriented network, compute a directed minimum u,t edge cut. We denote the set of edges of this cut as C_u.

These steps are then repeated for each $u \in V_r$. Letting

$$C_r := \bigcup_{u \in V_r} C_u, \tag{5.4}$$

we have the following.

Proposition 7 *For each shortest u, v-path in which $u, v \in V_r$, the edge set C satisfies at least one of the following two properties:*

1. *There is an edge $e \in C$ such that $e \in E_{uv}$.*

2. *There are edges $e_1, e_2 \in C$ such that $e_1 \in E_u$ and $e_2 \in E_v$.*

PROOF. By construction. ■

5.3.3 A 2-approximation proof

An obvious but important property of C_u for establishing the 2-approximation property of the iterative four-step algorithm is given in the following lemma.

Lemma 8 *Suppose that an edge set C is a solution to the OSP problem, then*

$$\sum_{e \in C \cap E'_u} c(e) \geq \sum_{e \in C_u} c(e). \tag{5.5}$$

PROOF. Every solution to the OSP problem must also solve the problem of predicting agents heading to vertex u. This part of the solution contains only edges from the set E'_u. Since the edge set C_u is an optimal solution for the sensor placement problem for predicting agents going to u, no other solution can have a lower cost. ■

Theorem 9 *Suppose that an edge set C is a solution to the OSP problem and C_r is the solution produced by the iterative four-step algorithm, then*

$$\sum_{e \in C} c(e) \leq \sum_{e \in C_r} c(e) \leq 2 \sum_{e \in C} c(e). \tag{5.6}$$

PROOF. By (5.4) and Lemma 8,

$$\sum_{e\in C_r} c(e) = \sum_{u\in V_r}(\sum_{e\in C_u} c(e)) \leq \sum_{u\in V_r}(\sum_{e\in C\cap E'_u} c(e)). \tag{5.7}$$

For each $u \in V_r$,

$$\sum_{e\in C\cap E'_u} c(e) = \sum_{e\in C\cap E_u} c(e) + \sum_{e\in C\cap(\cup_{u'\in V_r, u'\neq u} E_{uu'})} c(e) = \sum_{e\in C\cap E_u} c(e) + \sum_{u'\in V_r, u'\neq u}(\sum_{e\in C\cap E_{uu'}} c(e)). \tag{5.8}$$

We can then write the RHS of (5.7) as

$$\sum_{u\in V_r}(\sum_{e\in C\cap E_u} c(e) + \sum_{u'\in V_r, u'\neq u}(\sum_{e\in C\cap E_{uu'}} c(e))) = \sum_{e\in C\cap(\cup_{u\in V_r} E_u)} c(e) + \sum_{u\in V_r}(\sum_{u'\in V_r, u'\neq u}(\sum_{e\in C\cap E_{uu'}} c(e))). \tag{5.9}$$

Note that in the second three-level summation, each E_{uv}, $u, v \in V_r, u \neq v$, can appear at most twice. It is clear that $E_{uv} \cap E_{u'v'} = \varnothing$ when u, v, u', v' are all distinct; furthermore, $E_{uv} \cap E_{uv'} = \varnothing$. This means that the same $E_{uu'}$ will be looked at at most twice in the second summation in (5.9). That is (note that E_{uv} and E_{vu} are the same set),

$$\sum_{u\in V_r}(\sum_{u'\in V_r, u'\neq u}(\sum_{e\in C\cap E_{uu'}} c(e))) = \sum_{u,v\in V_r, u\neq v}(\sum_{e\in C\cap E_{uv}} c(e)) = \sum_{e\in C\cap(E\setminus(\cup_{u\in V_r} E_u))} c(e). \tag{5.10}$$

From here it is clear that the claim of the theorem holds. ■

REFERENCES

[1] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*, MIT Press, Cambridge, MA, 2005.

[2] J.-C. Latombe, *Robot Motion Planning*, Kluwer, Boston, MA, 1991.

[3] S. M. LaValle, *Planning Algorithms*, Cambridge University Press, Cambridge, U.K., 2006, also available at http://planning.cs.uiuc.edu/.

[4] G. E. Collins, Quantifier elimination by cylindrical algebraic decomposition–twenty years of progress, in *Quantifier Elimination and Cylindrical Algebraic Decomposition*, B. F. Caviness and J. R. Johnson, Eds., pp. 8–23. Springer-Verlag, Berlin, 1998.

[5] J. Davenport and J. Heintz, Real quantifier elimination is doubly exponential, *Journal of Symbolic Computation*, vol. 5, pp. 29–35, 1988.

[6] J. F. Canny, *The Complexity of Robot Motion Planning*, MIT Press, Cambridge, MA, 1988.

[7] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, *IEEE Transactions on Robotics & Automation*, vol. 12, no. 4, pp. 566–580, June 1996.

[8] S. M. LaValle, Rapidly-exploring random trees: A new tool for path planning, TR 98-11, Computer Science Dept., Iowa State University, October 1998.

[9] J. J. Kuffner and S. M. LaValle, RRT-connect: An efficient approach to single-query path planning, in *Proceedings IEEE International Conference on Robotics and Automation*, 2000, pp. 995–1001.

[10] S. Thrun, W. Burgard, and D. Fox, A probabilistic approach to concurrent mapping and localization for mobile robots, *Machine Learning*, vol. 31, no. 5, pp. 1–25, April 1998.

[11] D. Lieb, A. Lookingbill, and S. Thrun, Adaptive road following using self-supervised learning and reverse optical flow, in *Proceedings Robotics: Science and Systems*, 2005.

[12] A. Broggi, C. Caraffi, P. P. Porta, and P. Zani, The single frame stereo vision system for reliable obstacle detection used during the 2005 DARPA grand challenge on terramax, in *IEEE Intelligent Transportation System Conference*, 2006.

[13] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. M. Howard, S. Kolski, A. Kelly, M. Likhachev, M. McNaughton, N. Miller, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, B. Salesky, Y. Seo, S. Singh, J. Snider, A. Stentz, W. Whittaker, Z. Wolkowicki, J. Ziglar, H. Bae, T. Brown, D. Demitrish, B. Litkouhi, J. Nickolaou, V. Sadekar, W. Zhang, J. Struble, M. Taylor, M. Darms, and D. Ferguson, Autonomous driving in urban environments: Boss and the urban challenge, *Journal of Field Robotics*, vol. 25, no. 8, pp. 425–466, August 2008.

[14] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, D. Johnston, S. Klumpp, D. Langer, A. Levandowski, J. Levinson, J. Marcil, D. Orenstein, J. Paefgen, I. Penny, A. Petrovskaya, M. Pflueger, G. Stanek, D. Stavens, A. Vogt, and S. Thrun, Junior: The Stanford entry in the urban challenge, *Journal of Field Robotics*, vol. 25, no. 9, pp. 569–597, September 2008.

[15] J. Bohren, T. Foote, J. Keller, A. Kushleyev, D. Lee, A. Stewart, P. Vernaza, J. Derenick, J. Spletzer, and B. Satterfield, Little Ben: The Ben Franklin racing teams entry in the 2007 DARPA urban challenge, *Springer Tracts in Advanced Robotics*, vol. 56, pp. 231–255, 2009.

[16] D. E. Whitney, Real robots don't need jigs, in *Proceedings IEEE International Conference on Robotics & Automation*, 1986.

[17] J. van den Berg, S. Miller, K. Goldberg, and P. Abbeel, Gravity-based robotic cloth folding, in *Algorithmic Foundations of Robotics, IX*, David Hsu, Volkan Isler, Jean-Claude Latombe, and Ming C. Lin, Eds., vol. 68. Springer-Verlag, Berlin, 2011.

[18] M. Mason, Kicking the sensing habit, *AI Magazine*, vol. 14, no. 1, pp. 58–59, 1993.

[19] T. Başar and G. J. Olsder, *Dynamic Noncooperative Game Theory*, Academic, London, 2nd edition, 1995.

[20] H. W. Kuhn, Extensive games and the problem of information, in *Contributions to the Theory of Games*, H. W. Kuhn and A. W. Tucker, Eds., pp. 196–216. Princeton University Press, Princeton, NJ, 1953.

[21] D. P. Bertsekas, *Dynamic Programming: Deterministic and Stochastic Models*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[22] P. R. Kumar and P. Varaiya, *Stochastic Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[23] L. J. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, and R. Motwani, Visibility-based pursuit-evasion in a polygonal environment, *International Journal of Computational Geometry and Applications*, vol. 9, no. 5, pp. 471–494, 1999.

[24] J. Castellanos, J. Montiel, J. Neira, and J. Tardós, The SPmap: A probabilistic framework for simultaneous localization and mapping, *IEEE Transactions on Robotics & Automation*, vol. 15, no. 5, pp. 948–953, 1999.

[25] H. Choset and K. Nagatani, Topological simultaneous localization and mapping (T-SLAM), *IEEE Transactions on Robotics & Automation*, vol. 17, no. 2, pp. 125–137, 2001.

[26] G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba, A solution to the simultaneous localisation and map building (SLAM) problem, *IEEE Transactions on Robotics & Automation*, vol. 17, no. 3, pp. 229–241, 2001.

[27] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, FastSLAM: A factored solution to the simultaneous localization and mapping problem, in *Proceedings AAAI National Conference on Artificial Intelligence*, 1999.

[28] R. Parr and A. Eliazar, DP-SLAM: Fast, robust simultaneous localization and mapping without predetermined landmarks, in *Proceedings International Joint Conference on Artificial Intelligence*, 2003.

[29] F. V. Fomin, P. A. Golovach, A. Hall, M. Mihalák, E. Vicari, and P. Widmayer, How to guard a graph?, *Lecture Notes in Computer Science*, vol. 5369, pp. 318–329, 2008.

[30] Y. Ho, A. E. Bryson, and S. Baron, Differential games and optimal pursuit-evasion strategies, *IEEE Transactions on Automatic Control*, vol. 10, no. 4, pp. 385–389, 1965.

[31] R. Isaacs, *Differential Games*, Wiley, New York, 1965.

[32] O. Hájek, *Pursuit Games*, Academic, New York, 1975.

[33] L. Alonso, A. S. Goldstein, and E. M. Reingold, Lion and man: Upper and lower bounds, *ORSA Journal of Computing*, vol. 4, no. 4, pp. 447–452, 1992.

[34] M. Pachter, Simple motion pursuit-evasion differential games, in *Mediterranean Conference on Control and Automation*, Lisbon, Portugal, July 2002.

[35] S.-H. Lim, T. Furukawa, G. Dissanayake, and H. F. Durrant-Whyte, A time-optimal control strategy for pursuit-evasion games problems, in *Proceedings IEEE International Conference on Robotics & Automation*, 2004.

[36] V. Turetsky, Upper bounds of the pursuer control based on a linear-quadratic differential game, *Journal of Optimization Theory and Applications*, vol. 121, no. 1, pp. 163–191, April 2004.

[37] T. D. Parsons, Pursuit-evasion in a graph, in *Theory and Application of Graphs*, Y. Alavi and D. R. Lick, Eds., pp. 426–441. Springer-Verlag, Berlin, 1976.

[38] W.-P. Chin and S. Ntafos, Optimum watchman routes, *Information Processing Letters*, vol. 28, pp. 39–44, 1988.

[39] W.-P. Chin and S. Ntafos, Optimum watchman routes in simple polygons, *Discrete and Computational Geometry*, vol. 6, pp. 9–31, 1991.

[40] I. Suzuki and M. Yamashita, Searching for a mobile intruder in a polygonal region, *SIAM Journal on Computing*, vol. 21, no. 5, pp. 863–888, October 1992.

[41] S. M. LaValle, D. Lin, L. J. Guibas, J.-C. Latombe, and R. Motwani, Finding an unpredictable target in a workspace with obstacles, in *Proceedings IEEE International Conference on Robotics and Automation*, 1997, pp. 737–742.

[42] S. M. LaValle and J. Hinrichsen, Visibility-based pursuit-evasion: The case of curved environments, *IEEE Transactions on Robotics and Automation*, vol. 17, no. 2, pp. 196–201, April 2001.

[43] S.-M. Park, J.-H. Lee, and K.-Y. Chwa, Visibility-based pursuit-evasion in a polygonal region by a searcher, Tech. Rep. CS/TR-2001-161, Dept. of Computer Science, KAIST, Seoul, South Korea, January 2001.

[44] B. Simov, G. Slutzki, and S. M. LaValle, Pursuit-evasion using beam detection, in *Proceedings IEEE International Conference on Robotics and Automation*, 2000.

[45] T. Kameda, M. Yamashita, and I. Suzuki, On-line polygon search by a seven-state boundary 1-searcher, *IEEE Transactions on Robotics*, vol. 22, pp. 446–460, June 2006.

[46] B. P. Gerkey, S. Thrun, and G. Gordon, Visibility-based pursuit-evasion with limited field of view, *International Journal of Robotics Research*, vol. 25, no. 4, pp. 299–322, 2006.

[47] S. Sachs, S. Rajko, and S. M. LaValle, Visibility-based pursuit-evasion in an unknown planar environment, *International Journal of Robotics Research*, vol. 23, no. 1, pp. 3–26, January 2004.

[48] B. Tovar and S. M. LaValle, Visibility-based pursuit-evasion with bounded speed, *International Journal of Robotics Research*, 2007, Under review (invited submission from WAFR 2006).

[49] J. Sgall, A solution of David Gale's lion and man problem, *Theoretical Computational Science*, vol. 259, no. 1-2, pp. 663–670, 2001.

[50] S. Kopparty and C. V. Ravishankar, A framework for pursuit evasion games in R^n, *Information Processing Letters*, vol. 96, no. 3, pp. 114–122, 2005.

[51] S. D. Bopardikar, F. Bullo, and J. P. Hespanha, On discrete-time pursuit-evasion games with sensing limitations, *IEEE Transactions on Robotics*, vol. 24, no. 6, pp. 1429–1439, 2008.

[52] R. Vidal, O. Shakernia, H. J. Kim, D. H. Shim, and S. Sastry, Probabalistic pursuit-evasion games: Theory, implementation, and experimental evaluation, *IEEE Transactions on Robotics and Automation*, vol. 18, no. 5, pp. 662–669, 2002.

[53] V. Isler, S. Kannan, and S. Khanna, Randomized pursuit-evasion in a polygonal environment, *IEEE Transactions on Robotics*, vol. 5, no. 21, pp. 864–875, 2005.

[54] S. Alexander, R. Bishop, and R. Ghrist, Pursuit and evasion in nonconvex domains of arbitrary dimension, in *Robotics: Science and Systems II*, G. S. Sukhatme, S. Schaal, W. Burgard, and D. Fox, Eds. MIT Press, Cambridge, MA, 2007.

[55] J. Aslam, Z. Butler, F. Constantin, V. Crespi, G. Cybenko, and D. Rus, Tracking a moving object with a binary sensor network, in *1st International Conference on Embedded Networked Sensor Systems*, 2002, pp. 150–161.

[56] W. Kim, K. Mechitov, J. Choi, and S. Ham, On target tracking with binary proximity sensors, in *ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2005, pp. 301–308.

[57] N. Shrivastava, R. Mudumbai, U. Madhow, and S. Suri, Target tracking with binary proximity sensors: Fundamental limits, minimal descriptions, and algorithms, in *Proceedings 4th International Conference on Embedded Networked Sensor Systems*, 2006, pp. 251–264.

[58] J. Singh, R. Kumar, U. Madhow, S. Suri, and R. Cagley, Tracking multiple targets using binary proximity sensors, in *Proceedings Information Processing in Sensor Networks*, 2007.

[59] Y. Baryshnikov and R. Ghrist, Target enumeration via integration over planar sensor networks, in *Proceedings of Robotics: Science and Systems IV*, Zurich, Switzerland, June 2008.

[60] Q. Fang, F. Zhao, and L. Guibas, Counting targets: Building and managing aggregates in wireless sensor networks, Tech. Rep. P2002-10298, Palo Alto Research Center, June 2002.

[61] M. Dehn, *Papers on Group Theory and Topology*, Springer-Verlag, Berlin, 1987.

[62] D. B. A. Epstein, M. S. Paterson, G. W. Camon, D. F. Holt, S. V. Levy, and W. P. Thurston, *Word Processing in Groups*, A. K. Peters, Natick, MA, 1992.

[63] B. Tovar, F. Cohen, and S. M. LaValle, Sensor beams, obstacles, and possible paths, in *Proceedings Workshop on Algorithmic Foundations of Robotics*, 2008.

[64] E. Asarin, O. Bournez, T. Dang, and O. Maler, Approximate reachability analysis of piecewise-linear dynamical systems, in *Hybrid Systems: Computation and Control*. 2000, pp. 21–31, Springer.

[65] P. Cheng and V. Kumar, Sampling-based falsification and verification of controllers for continuous dynamic systems, in *Proceedings Workshop on Algorithmic Foundations of Robotics*, 2006.

[66] A. Chutinan and B. Krogh, Computational techniques for hybrid system verification, *IEEE Transactions on Automatic Control*, vol. 48, no. 1, pp. 64–75, January 2003.

[67] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas, Discrete abstractions of hybrid systems, in *Proceedings of the IEEE*, 2000, pp. 971–984.

[68] N. Piterman, A. Pnueli, and Y. Sa'ar, Synthesis of Reactive(1) designs, in *Proceedings Verification, Model Checking, and Abstract Interpretation*, 2006, pp. 364–380.

[69] D. C. Conner, H. Kress-Gazit, H. Choset, A. A. Rizzi, and G. Pappas, Valet parking without a valet, in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, 2007.

[70] H. Kress-Gazit, G. E. Fainekos, and G. Pappas, Where's Waldo? Sensor-based temporal logic motion planning, in *Proceedings IEEE International Conference on Robotics & Automation*, 2007, pp. 3116–3121.

[71] J. Yu and S. M. LaValle, Tracking hidden agents through shadow information spaces, in *Proceedings IEEE International Conference on Robotics & Automation*, 2008, pp. 2331–2338.

[72] J. Yu and S. M. LaValle, Probabilistic shadow information spaces, in *Proceedings IEEE International Conference on Robotics & Automation*, 2010, pp. 3543–3549.

[73] J. Yu and S. M. LaValle, Shadow information spaces: Combinatorial filters for tracking targets, *IEEE Transactions on Robotics*, vol. 28, no. 2, pp. 440–456, February 2012.

[74] J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press, New York, 1987.

[75] T. Shermer, Recent results in art galleries, *Proceedings of the IEEE*, vol. 80, no. 9, pp. 1384–1399, September 1992.

[76] J. Yu, S. M. LaValle, and D. Liberzon, Rendezvous without coordinates, *IEEE Transactions on Automatic Control*, vol. 57, no. 2, pp. 421–434, February 2012.

[77] Y. Baryshnikov and R. Ghrist, Target enumeration via Euler characteristic integrals, *SIAM Journal of Applied Mathematics*, vol. 70, no. 4, pp. 825, August 2009.

[78] B. Gfeller, M. Mihalak, S. Suri, E. Vicari, and P. Widmayer, Counting targets with mobile sensors in an unknown environment, in *ALGOSENSORS*, July 2007.

[79] C.-C. Wang, C. Thorpe, S. Thrun, M. Hebert, and H. Durrant-Whyte, Simultaneous localization, mapping and moving object tracking, *International Journal of Robotics Research*, vol. 26, no. 9, pp. 889–916, 2007.

[80] D. B. Yang, H. H. Gonzalez-Banos, and L. J. Guibas, Counting people in crowds with a real-time network of simple image sensors, in *Proceedings IEEE International Conference on Computer Vision*, 2003, vol. 1, pp. 122– 129.

[81] L. D. Stone, *Theory of Optimal Search*, Academic Press, New York, NY, 1975.

[82] I. Yan and G. L. Blankenship, Numerical methods in search path planning, in *Proceedings of the 27th IEEE Conference on Decision and Control*, 1988, vol. 2, pp. 1563–1569.

[83] R. Mahler, Objective functions for Bayesian control-theoretic sensor management, II: MHC-like approximation, in *New Developments in Cooperative Control and Optimization*, S. Butenko, R. Murphey, and P. Paralos, Eds., Norwell, MA, 2003, pp. 273–316, Kluwer Academic Publishers.

[84] F. Bourgault, T. Furukawa, and H.F. Durrant-Whyte, Coordinated decentralized search for a lost target in a Bayesian world, in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003, pp. 48–53.

[85] P. E. Berry, C. Pontecorvo, and D. A. B. Fogg, Optimal search, location and tracking of surface maritime targets by a constellation of surveillance satellites, Tech. Rep., DSTO Information Sciences Laboratory, Edingburgh, SA, 2003.

[86] S. Petitjean, D. Kriegman, and J. Ponce, Computing exact aspect graphs of curved objects: Algebraic surfaces, *International Journal of Computer Vision*, vol. 9, pp. 231–255, December 1992.

[87] B. Tovar, R Murrieta, and S. M. LaValle, Distance-optimal navigation in an unknown environment without sensing distances, *IEEE Transactions on Robotics*, vol. 23, no. 3, pp. 506–518, June 2007.

[88] J. O'Rourke, Visibility, in *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, Eds., pp. 643–663. Chapman and Hall/CRC Press, New York, 2nd edition, 2004.

[89] M. Pocchiola and G. Vegter, The visibility complex, *International Journal Computational Geometry & Applications*, vol. 6, no. 3, pp. 279–308, 1996.

[90] C. H. Papadimitriou and K. J. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[91] A. Schrijver, *Combinatorial Optimization*, Springer-Verlag, 2003.

[92] R. E. Kalman, A new approach to linear filtering and prediction problems, *Transactions of the ASME, Journal of Basic Engineering*, vol. 82, pp. 35–45, 1960.

[93] J. Edmonds and R. M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of the ACM*, vol. 19, no. 2, pp. 248–264, 1972.

[94] P. Bose, A. Lubiv, and J. I. Munro, Efficient visibility queries in simple polygons, in *Proceedings Canadian Conference on Computational Geometry*, 1992, pp. 23–28.

[95] A. V. Goldberg and R. E. Tarjan, A new approach to the maximum flow problem, in *STOC '86: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 1986, pp. 136–146, ACM.

[96] S. M. LaValle, Filtering and planning in information spaces, Tech. Rep., Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 2009.

[97] R. A. Wagner, Order-n correction for regular languages, *Communications of the ACM*, vol. 17, no. 5, pp. 265–268, 1974.

[98] R. A. Wagner and J. I. Seiferas, Correcting counter-automaton-recognizable languages, *SIAM Journal on Computing*, vol. 7, no. 3, pp. 357–375, August 1978.

[99] D. Sankoff and J. B. Kruskal, *Time Wraps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, 1983.

[100] E. W. Myers and W. Miller, Approximate matching of regular expressions, *Bulletin of Mathematical Biology*, vol. 51, no. 1, pp. 5–37, 1989.

[101] C. Allauzen and M. Mohri, Linear-space computation of the edit-distance between a string and a finite automaton, in *London Algorithmics 2008: Theory and Practice*, 2008.

[102] G. Navarro, A guided tour to approximate string matching, *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[103] J. Yu and S. M. LaValle, Cyber detectives: Determining when robots or people misbehave, in *The Ninth Workshop on Algorithmic Foundations of Robotics*, 2010, pp. 391–407.

[104] J. Yu and S. M. LaValle, Story validation and approximate path inference with a sparse network of heterogeneous sensors, in *Proceedings IEEE International Conference on Robotics & Automation*, 2011, pp. 4980–4985.

[105] R. E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

[106] B. Chazelle, Approximation and decomposition of shapes, in *Algorithmic and Geometric Aspects of Robotics*, J. T. Schwartz and C. K. Yap, Eds., pp. 145–185. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[107] J. Hopcroft and R. Tarjan, Efficient algorithms for graph manipulation, *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[108] L. R. Ford and D. R. Fulkerson, Maximal flow through a network, *Research Memorandum RM-1400, The RAND Corporation*, November 1954.

[109] N. Garg, V. V. Vazirani, and M. Yannakakais, Multiway cuts in directed and node weighted graphs, in *Proceedings of the 21st International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 820*. 1994, pp. 487–498, Springer-Verlag.

[110] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, The complexity of multiterminal cuts, *SIAM Journal on Computing*, vol. 23, pp. 864–894, 1994.